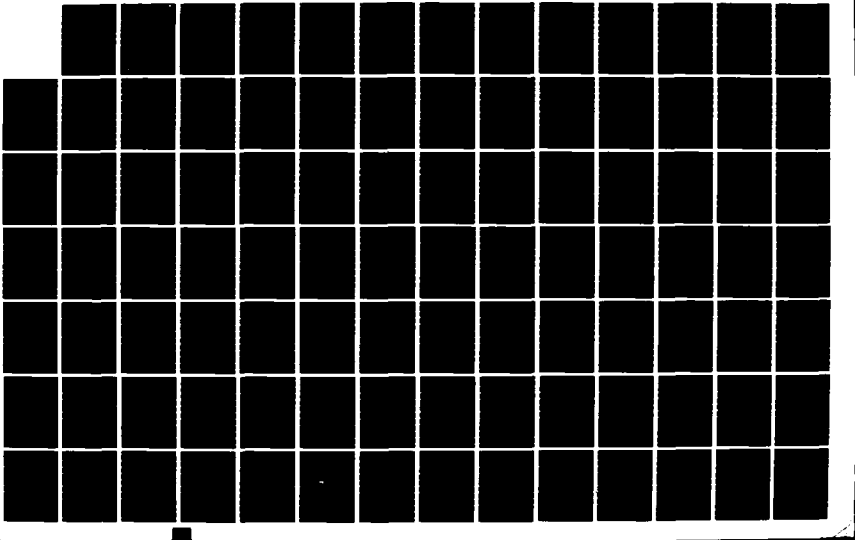
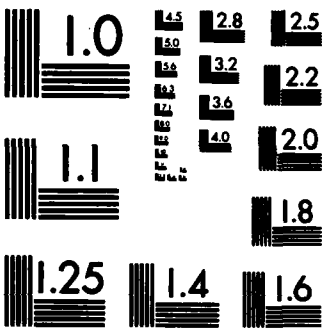
AD-A132 569 FORMAL TECHNIQUES IN THE MANAGEMENT OF SOFTWARE DESIGN 1/4
(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH
W E RICHARDSON 17 JUN 83 AFIT/CI/NR-83-28D

UNCLASSIFIED F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

UNCLASS

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: AFIT/CI/NR 83-28D	2. GOVT ACCESSION NO: AD-A132569
3. RECIPIENT'S CATALOG NUMBER:	
4. TITLE (and Subtitle): Formal Techniques in the Management of Software Design	5. TYPE OF REPORT & PERIOD COVERED: THESIS/DISSERTATION
	6. PERFORMING ORG. REPORT NUMBER:
7. AUTHOR(s): William E. Richardson	8. CONTRACT OR GRANT NUMBER(s):
9. PERFORMING ORGANIZATION NAME AND ADDRESS: AFIT STUDENT AT: Oxford University	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:
11. CONTROLLING OFFICE NAME AND ADDRESS: AFIT/NR, WPAFB OH 45433	12. REPORT DATE: 17 June 1983
	13. NUMBER OF PAGES: 306
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office):	15. SECURITY CLASS. (of this report): UNCLASS
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report)

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

APPROVED FOR PUBLIC RELEASE: IAW AFR 190-17

1 SEP 1983

Approved for public release: IAW AFR 190-17.

LYNN E. WOLAVER

Dean for Research and Professional Development

Air Force Institute of Technology (ATC)

Wright-Patterson AFB OH 45433

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

ATTACHED

DTIC ELECTE SEP 16 1983 H

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE UNCLASS

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AD A132569

DTIC FILE COPY

83 09 14 090

Formal Techniques

in the

Management

of

Software Design

Accession For	
NTIS GRA&I	☑
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A	

by

William E. Richardson

Merton College

June 17, 1983

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy in the Faculty of Mathematics at the University of Oxford.

ABSTRACT

Formal Techniques in the Management of Software Design

William E. Richardson

Merton College

Doctor of Philosophy

Trinity Term 1983

The inordinately high cost of software continues to be the major shortcoming in the development of computer systems. In the past, attempts to solve this software crisis have been from one of three independent approaches -- using structuring techniques or using formal techniques (together these two are called software engineering) or using management techniques. It is now apparent that this management-technology decoupling is avoidable and that a viable software design methodology must include mutually supportive management, structuring, and formal components. This thesis attempts to develop just such a methodology for the design of large systems.

~~First, we~~ propose a set of criteria which will be use to evaluate design methodologies. Based on these criteria and research into existing methodologies, ~~we~~ then outlines ~~our~~ new methodology. It utilizes the advantages of high level abstraction, an extensible set theoretical notation, hierarchical structuring, and numerous management techniques. A simple example is given to introduce the design style and notation.

In order to assess the new methodology and its interaction of management and software engineering techniques, ~~we do~~ a case study development of a windowed, information sharing display and filing system. This development has as its starting point a simple but very powerful abstraction which could be used as the basis for any similar system. The abstraction is presented with an appropriate natural language explanation so that it could be a self contained entry in a library of high level abstractions.

On the basis of the case study, we give a partial assessment of the methodology against the stated criteria. Two existing methodologies are also assessed for comparison.

ACKNOWLEDGEMENTS

For all manner of support, including clerical and morale, I thank my wife Cynthia. Without her diligent efforts and indulgence this work would certainly never have come to fruition.

I would also be remiss if I did not mention my parents to whom I generally owe so much but who seldom get the appropriate expression of thanks.

For his constant inspiration and succor, I wish to express my thanks to my supervisor, Professor C. A. R. Hoare. His comments were always thought provoking and relevant.

My thanks also to Alex Teruel and Terry Erdle for the interesting discussions and for their perspectives on the problems of the world and computer science.

Finally, I wish to express my graditude to those who made my efforts financially possible: The Committee of Vice-Chancellors and Principals of the Universities of the United Kingdom who selected me for an Overseas Research Student Grant for each of my last two years; and the United States Air Force, the United States Air Force Academy Department of Computer Science, and the Air Force Institute of Technology who collectively afforded me all other material support. To these organizations I give my sincere thanks.

I dedicate this research to my beautiful new daughter, Suzanne; may she too discover the excitement that learning can bring.

TABLE OF CONTENTS

CHAPTER I

Introduction

Today's major shortcoming in the development of computer systems is, just as it was ten years ago, the cost of the software. Far from having been solved, this problem appears to be growing more critical as the proliferation of computer systems increases at an unprecedented rate. It is, therefore, imperative that we strive to define and implement a practical software development methodology which consolidates the technical and management gains we have made in the software development field. The overall aim of this thesis is to contribute to the design of just such a methodology. In particular, we investigate the practical application and management of formal software techniques within a development methodology for large scale sequential software systems.

I.1 The Problem.

At present, it is estimated that 75% of the purchase price of an off the shelf computer system is allocated to the cost of the software[Boehm,81]. Often more than 50% of this software expense comes from the cost of maintenance (either repair or enhancement.) The decade from 1980 to 1990 will see (in the US, for example) a six-fold increase in the computer and information processing industry[Boehm,81]. This means that at the current cost ratios, the US in 1990 will be spending about $120,000,000,000 per year on software maintenance. However, as hardware becomes economically feasible for more applications and as the scarcity of trained personnel intensifies, overall software costs are very likely to increase significantly faster than even this prediction would indicate[Wasserman,81; Distaso,80]. It is also likely that the percentage of the software budget consumed by maintenance will increase to as much as 80% in this decade[Pressman,82].

Of course, only the *direct* costs of software development are defined in these statistics. Of equal or greater importance are the *indirect* costs. These result from loss of business due to late delivery or software failure, disruption of the client's organization, system misuse or difficulty of use, system failure to keep user's trust, etc. Although these costs are of great significance they cannot, in general, be accurately measured.

The cost of software will continue to be the weak link in the computer system development chain and may even retard the growth in computer applications if it continues to climb. Stabilization and eventual decrease of overall cost requires a corresponding increase in software development productivity, reliability, and ease of adaptation, along with an increased concern for the indirect costs attributable to software production and use.

I.2 The Solution.

a. History.

Previous attempts to deal with the crucial problem of software cost have been numerous and varied. In general, these cost limiting strategies fall into one of two approaches -- software engineering or management. This separation of approaches has been called the "management-technology decoupling." [Boehm,76]

For our purposes, software engineering can be defined as the application of scientific ('engineering-like') principles to any stage of software design and maintenance. Two general forms of software engineering techniques have been proposed -- structuring techniques, which are processes to guide and delimit the software during development, and formal techniques, which are well formulated scientific or mathematical processes involving the use and manipulation of meaningful symbols. In general, the software engineering approach has the goal of improving the readability, verifiability, reliability, and maintainability of the software.

Structuring techniques are based on the general theory that "the structure of the program itself is the single most important determinant of the lifecycle costs of a software project." [Bergland,81;iii] These techniques attempt to enforce a division of labor, allow informal reasoning about program parts, and allow the substitution of equivalent modules within a program.

A few of the structuring techniques which have been applied with success to different aspects of the software development process are:

decomposition/modularization	[Parnas.72]
goto-less programming	[Dijkstra.68]
structured programming	[Dahl.72]
stepwise refinement	[Wirth.71].

The formal techniques suggest a two part approach to the problem: the first part is production of the design *specification*, the statement that formally embodies the system requirements and defines the behavior of the system. The second part is the formal *verification* that the implemented software rigorously adheres to the specification of the design.

"The formal methods seek to do for programming what mathematics has done for engineering: provide symbolic methods whereby the attributes of an artifact can be described and predicted." [Berg.82;1]

One or more symbolic notations and the mathematical methods to manipulate the notation must be incorporated into any viable formal technique. (See [Goguen.80] or [Berg.82] for a further discussion of what is desirable in such a formal symbology.) A few of the representations which have been proposed for use in formal techniques include:

CLEAR	[Burstall.80]
set theory	[Abrial.82]
SPECIAL	[Levitt]
temporal logic	[Manna.80]
algebraic specifications	[Horning.80]
META-IV	[Bjorner.82]

The management approach adheres to the philosophy that general and applied management practices are effective for controlling software development and its associated cost. Proponents of this approach stress that "it is fundamentally a management problem to insure that the product will in fact be what the user wants." [Cave.78] Often, unlike the software engineering approach, management strategies will focus on the software development and maintenance process itself rather than the resulting software. That is, with the management approach, cost reduction occurs generally as a result of a simplified or more efficient development process rather than

the development of a superior implementation. Experience has shown, however, that a simplified development technique may also have the secondary effect of promoting better programs. Although the management approach is by far the most varied and arguably the most successful of the current attempts at moderating software costs, it has become obvious that there are no purely management solutions to technical problems. A few of the numerous management strategies which have been practically applied are:

egoless programming	[Weinberg,69]
chief programmer teams	[Baker,72]
unit development folders	[Ingrassia,78]
walk-throughs	[Yourdon,78].

In conjunction with these approaches to minimizing software costs, automated tools and program development environments have been created. Such tools are generally designed to effect a savings by simplifying and enforcing either a software engineering or a management strategy. To date, the achievements of such tools have been limited as a result of two serious shortcomings[Wasserman,82]:

1. They fail to support a software development methodology or assist in the control of a software development process.

2. They fail to support the entire software development lifecycle.

That is, such tools have been only marginally successful because they are very isolated in terms of their scope and integration into the overall development process.

This brings us to a second standard by which we can categorize software cost reduction strategies; that is, the extent of the software lifecycle over which the strategy is effective[Berg]. A <u>method</u> is a strategy which directly affects the software, design, or the development process in a restricted portion of the lifecycle. A strategy or coordinated group of strategies which consistently influence a significant portion of the lifecycle is called a <u>methodology</u>. By far, the larger number of strategies introduced have been methods designed to influence one phase of the lifecycle, usually

implementation. However, a few methodologies have been defined; for example:

top-down design methodology	
Jackson structured design methodology	[Jackson,75]
chief programmer teams	[Baker,72]
Hierarchical Development Methodology (HDM)	[Silverberg]
Rigorous Methodology	[Jones,80]
Rational Design Methodology (RDM)	[Boyd,a]
USAF Methodology	[USAF]

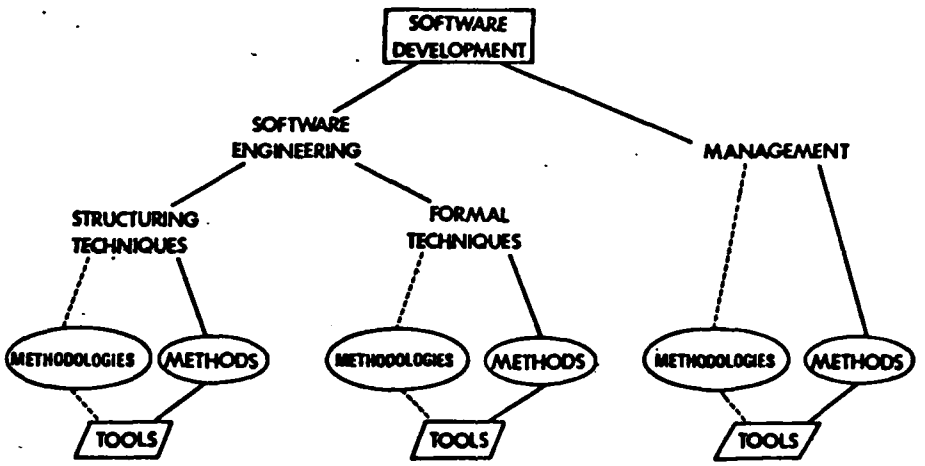


FIG. I-1 -- Historical Approaches to the Software Development Problem.

b. Our Approach.

The approach taken in this thesis is that the gap between the software engineering and management strategies is avoidable and can be bridged with a synergistic result. Further, such a combined strategy is required if the critical problem of software cost is to be effectively contained in the design and development of large scale application software systems. Otherwise, the problems of communication, management control and correctness of the product grow in a geometric relation to the size of the project.[Brooks,75]

In order to design a practical combined software engineering-management approach, we have adapted or developed selected formal, structuring, and management methods and have composed them into an integrated whole. The resultant methodology combines many desirable cost reducing features of the previously independent approaches. These features include:

- use of very abstract initial designs and potential reuse of previous abstraction designs
- expanded set theoretical formal design notation which allows formal verification and testing of design features
- separation of formal verification and design for increased management control
- vertical and horizontal decomposition/recomposition to "divide and conquer" the design problem
- separation of design and implementation decisions
- a limitative design approach to constrain the complexity of the design
- use of baselines, milestones and configuration management controls within the formal design process
- use of prototypes
- progressive requirement development with continuous client involvement
- continuous design documentation with an explicit statement of design decisions

In general, this approach to the design of large software systems attempts to insure that "everyone will know what they ought to know when they ought to know it."

I.3 Thesis Organization.

In Chapter II we define a set of important criteria which we will use to assess software development methodologies. We then develop the main points of our proposed methodology, giving emphasis to the interaction among the formal, structural, and management aspects of the technique. To conclude

the second chapter we present a simple introductory example of the formal design structure and notation. In the third chapter we define a very abstract family of information sharing filing systems. This family is also presented in the style and notation of the proposed methodology and forms the basis of one component in the case study design of Chapters IV and V. Our case study is the design of a windowed, information sharing, visible filing system. It demonstrates an application of the methodology to a reasonably sized design problem. The complexity of this development has been controlled by eliminating most communication requirements (ie. a one man design and implementation with a hypothetical client) and by concentrating on the unique components of the methodology during the first half of the software life cycle.

In Chapter VI we evaluate the case study application of the methodology and extrapolate these conclusions to large scale, multi-man developments over the full lifecycle. By so doing, we prepare the way for more substantial tests of the methodology. Additionally, in this chapter we compare the proposed development technique with two other techniques of interest: the methodology currently used by the US Air Force and the Hierarchical Development Methodology of SRI International.

The final chapter summarizes the significance of the proposed methodology as an approach to the problem of software cost and reviews other contributions of this research. For completeness, a sketch of possible future research is given.

CHAPTER II

Methodology

Although the historical approaches listed in the previous chapter gave us some general guidelines, substantial further research was required to produce a practical methodology based on the experience gained from those approaches. We surveyed the significant existing software design techniques to determine if, and how, they attempted to satisfy the problems of large scale software system development. Then we defined the most important criteria by which to judge any proposed methodology. These criteria are presented in the following section. From our research and criteria definitions, we produced a set of basic tenets which appear to collectively define an improved software development methodology for large systems. Section II.2 contains these tenets in the form of an overview of the important aspects of our methodology. The reader is advised that the recommended approach to Section II.2 is an initial quick skim, followed by a more detailed perusal, if required, after reading the methodology overview in Section II.3 and the small example application in Section II.4.

II.1 Criteria.

Each of the historical approaches and techniques discussed in Chapter I provides a partial solution to the software development problem; likewise, each lacks some advantage provided by the others. Therefore, in a sense, our general criteria for a more effective software development strategy must include a capability for structuring, verifying, and managing the development process consistently throughout the lifecycle of the software product. However, these three historical strategies are much too specific

and are occasionally even orthogonal to the actual characteristics we require to meet our stated goal of improved software development. A number of other attempts to develop criteria for the comparison and evaluation of software design methodologies have produced other general criteria. Some of these criteria can be found in [Berg.80; Boyd.a; Carlson.82; Keeton.80; Griffiths.78; Parker.78; Peters.79; Pressman.82], etc. In all cases, the criteria presented by these authors do not put the emphasis in the area of software cost reduction or are too general to be of use to us. Consequently, we have formulated our own criteria to emphasize the key areas as determined by our research. It would be impossible to present a *complete* set of criteria so we have limited ourselves to the more significant ones.

a. Control of Creativity And Cumulativity.

As in any design process, software design is a combination of two activities -- creation (innovation) and reapplication of previously designed components or reuse of concepts. (The example of architectural design often comes to mind when discussing the design process. Certainly, it is easy to relate these two activities to the design of a building or a bridge.) The first activity is highly dependent upon the utilization of human intelligence and cannot generally be distilled down to a set of step-by-step procedures. Consequently, this is an unpredictable process which is subject to error, backtracking, and multiple iterations, and is difficult to manage. Therefore, the creative nature of design cannot be without restraint, and must, in the end, lead to the objectives of correctness, completeness, robustness, maintainability, adaptability, economy, etc. At the same time, this restraint cannot be guilty of stifling the creative ability of the software engineer either by overly complex notation or by drowning him in a morass of detail.

The second activity of design requires previous designs be available and modifiable to fit new circumstances. This capability for reapplication design efforts to new but similar problems could be called the <u>cumulativity</u> of a design field. The danger inherent in not requiring a design process to be cumulative is that errors are repeated, lessons are not learned through experience, and solutions do not improve even though the problems get more complex or critical. At present, in spite of vigorous pursuit of this objective, the cumulativity of software design has remained at a very low level[Branstad.81].

Therefore, a software design methodology should provide:

1. a technique for identifying and isolating components which must be creatively designed.

2. an environment which enhances creativity by guiding and structuring but does not overwhelm the creative effort.

3. a technique for expanding the reapplication of designs (not just programs[Parnas.75])

b. Guaranteed Product Reliability.

An important responsibility of any design methodology is to *adequately* guarantee the reliability of the design and its implementation. Put slightly differently, the designer should, to the requisite level, *validate* the accuracy of the design's representation of the client's requirements and *verify* that implementations accurately correspond to the designs. It is significant to note that the communication of requirements from the client to the designer is the single link in the development process which cannot be formalized; hence, validation will always remain a subjective and informal task.

Verification can be either informal or formal. Informal verification is an attempt to find and correct errors early in the design and implementation stages. Formal verification works at proving the mathematical correctness of the implementations in a formal notation. However, totally formal techniques are expensive to the point of being impractical for use on large developments[Berg.82; Pressman.82]. The true advantage in reliability afforded by the use of formal specifications is in the discipline it imposes so that programs need not be proven correct so much as developed in such a manner as to make their correctness evident[Dijkstra.76].

Our criterion of guaranteed product reliability does not attempt to dictate absolute correctness of the design and implementation, which would be an unrealistic standard. Instead, it requires that management can prescribe and control the level of reliability for individual system components. Consequently, within the scope of a methodology the development manager should be able to dictate the rigor[Jones.80] of the verification and (within limits) the granularity of the specification. From this we can see that formal and management framework of a methodology must be mutually supportive and cannot just be allowed to coexist.

c. Requirement Evolution.

"The problem of generating complete, consistent, unambiguous, testable software requirements continues to plague the software industry and remains the most serious challenge to bring order to the software development process.... This problem, which has been debated, analyzed, and researched for years, is invariably the critical factor in project success or failure for most projects of significant size. [One of] the key elements of this problem ... [is] continually changing user needs." [Distaso,80; 10]

This quotation sums up well the critical and difficult nature of requirements definition in the software development process. And it suggests why, in any design process, requirements definition must be accomplished in an evolutionary and iterative manner. It must be evolutionary because requirements are usually hierarchical -- one does not usually stipulate the size of the master bedroom before defining the style of the house he wishes the architect to design. Each new level of requirement builds on the preceding levels of requirement definition; at each level the client wants to be left with some freedom for expressing his requirements as the system takes shape. It is an iterative process because backtracking to a previous level may be required when it becomes obvious that a new requirement is contradictory to those previously defined. This view of requirements definition as evolutionary and iterative leaves one wondering how it differs from the process of hierarchical design. As the design theorist Rittel stressed: "A statement of the problem is a statement of the solution."[Rittel,73] If the design problem statement and the design solution statement are in the same terms (ie. both formal, both in natural language text, both in graphical representation, etc.) they are essentially equivalent statements. It is, therefore, not only unnatural but counterproductive to attempt to separate formal requirements definition and the formal design process.

The client, unless he is highly sophisticated or has a trivial problem, is likely to have only a limited initial understanding of his requirements and will undoubtedly express those requirements in an incomplete (and often inconsistent) manner in natural language. For every set of circumstances there is a different requirement interpretation for each different client (and a different design interpretation of the requirement definition for each different designer.)

This wide variation in client requirement interpretation impels any design methodology which starts with an informal requirement definition (as invariably it must) to allow great flexibility in its formal starting point. The

key, then, in the difficult transition from the incomplete informal requirements definition to a formal definition (ie. a formal design) is to gain confidence in the formal translation slowly, in an *evolutionary* manner with provision for *iteration*.

d. Continuity Through the Lifecycle.

Although most of today's software cost comes from the maintenance end of the lifecycle, it has often been suggested (eg. [Buckle,77]) that a remedy to this situation will require a concentrated effort in all phases of software development and in the early phases in particular. Hence, any viable methodology must cover the entire life of the software system but must also encourage early identification of critical factors in the design process.

Most authors include some combination similar to the following in a description of the software lifecycle: problem definition, requirements specification, design, implementation, test, installation, maintenance/modification, and phase out. However, the separation between the various lifecycle phases is not always distinct; and this is especially true between requirements/design [Peters,78; Riddle,78], and design/implementation [Swartout,82]. Optimally, a software engineering methodology will not only consider the entire lifecycle, but will also reflect the similarity of consecutive lifecycle phases. That is, if, in a practical sense, design is indistinguishable from requirements specification, then the techniques, tools, languages, etc. used in the methodology should be the same for both of these phases.

In general, a verifiable software system, throughout its lifecycle, will have at least three different representations. The first is the natural language (or at least, non-formal) representation of the client's requirements. The second representation is the formal and verifiable language of specification. The final representation is the machine executable one -- the software itself. (The first two representations are required as the documentation of the third.) Consequently, it is obvious that at least two transitions will be required in a methodology which attempts to deal with the entire software lifecycle. Insuring smooth transitions among these representations is one of the major tasks of any methodology.

In sum, a viable methodology must reduce system lifecycle costs by early recognition of potential design difficulties and by comprehensive enforcement of the appropriate design procedures throughout the life of the software system. These procedures must reflect the life stages of the software and, further, must minimize the disruptive effect of representation changes.

e. Continuous Client Involvement.

Yet another vital element must be considered in the criteria for a good design methodology. The client (user, customer, etc.) is the driving force behind a system design and, hence, the supreme judge of any software effort. He will invariably look at a newly developed system and ask: "Did I get quality for the cost?" A design methodology must attempt to create a relationship between the client's perception of the design quality and the design cost which weighs heavily on the side of quality. This cannot be done formally but experience shows that the best way to succeed in this task is to keep the client actively involved from beginning to end -- it must be an interactive process between the client and the designer. The psychological benefits (acceptence, increased satisfaction, feeling of contributing)[Shneiderman,80] and the actual benefits (reduced training, reduced modification, etc.) from client involvement can be the difference between success and failure of a software project.

As we noted in the section on *Requirements Evolution*, the client's view of the problem is not static. This point is also succinctly presented in a quote from [Peters,78]:

"...Although the customer may state his requirements very firmly at the beginning, his perception of the problem begins to change as he begins to consider how the solution development (ie. design and perhaps coding) is proceeding."

Evolving requirements are the rule rather than the exception in large scale application software development. Obviously, it is important that design activity not be allowed to advance faster than the client's requirements evolve.

Our conclusion is that client interaction must be an integral part of the design methodology. This communication must be continuous, and it must occur in both directions. And finally, in order to keep the design in concert with the client's view of the requirements, the methodology must permit the judicious postponement of design decisions.

f. Manageability.

Because of the complexity inherent in the development of large scale software systems, the ability to manage that complexity within the framework of a methodology is mandatory. For our purposes, we have categorized the

areas which most require inherent manageability within the discipline of the methodology as:

1. visibility and control
2. budgeting and accountability
3. coordination and communication.

The first area, visibility and control, is at the heart of process management for any process. Visibility means that the manager can accurately determine the status of the design effort at critical points. Control indicates that based on the current status of the project he can direct the project in the manner which best suits his predetermined goals. In other words, the methodology must give the design manager the flexibility to select the most appropriate compromise from the conflicting design goals; it must give him the capability to determine if the design is progressing toward the compromise he has selected; and finally, it must give him the capability to redirect the design effort as necessary to achieve those goals.

The manageability of the resource budgeting and accounting in a development process is also significantly dependent upon the visibility of the process. In general, resource control in a process requires an initial estimate of resource allocation (budgeting) and periodic reviews (accounting) of resource usage followed by estimate refinement. A methodology which does not provide the capability to accurately determine resource allocations and allow timely review of resource utilization will often prove dangerously uncontrollable.

The final area of required manageability within a methodology is in communication and coordination. As noted in that classic of the software management field, [Brooks,75], the larger the project, the more difficult communication and coordination becomes. We previously stressed the importance of communication with the client. Also of significance is management control over the communication and coordination within the development team (designers, implementors, verifiers, maintainers, etc.) A major concern here, of course, is with the style and standard of the documentation. The ultimate answer to the problem of documentation is that the working of the methodology should *automatically* create the documentation in the form required without management intervention. Especially important in the documentation would be the explicit declaration of decisions and their rationale.

In summary, the criterion of manageability suggests that a design methodology, no matter how complete otherwise, is not acceptable unless it allows (or better still, enforces) the process control required by management. A methodology must view the software development process as a whole, not just program production but also budgets, schedules, priorities, operations, etc.

Again, it should be emphasized that these criteria represent a minimum set of capabilities we feel are desirable in a practical and viable software design methodology in order to effectively reduce the cost of software development. These criteria are obviously not mutually exclusive and in some instances may even represent overlapping or conflicting goals; therefore, the objective of determining a methodology to meet these criteria is an inherently complex task. As [Boehm,81:23] suggests,

"The most important software engineering skills we must learn are skills involved in dealing with a plurality of goals which may be at odds with each other, and the skill of coordinating the application of a plurality of means, each of which provides a varying degree of help or hindrance in achieving a given goal."

II.2 Software Engineering Basis.

We introduce the technical concepts which form the software engineering basis of our methodology in the following subsections. Standard management concepts will not be explained in detail since they can be referenced in any good software management book (eg. [Jensen,79; Donaldson,78]). In the Section II.3 we integrate the software engineering and management aspects into a complete design system.

a. Initial abstraction.

Initial requirements definitions will vary greatly in completeness and stability depending on many factors outside the control of the software engineer. Perhaps even more distressing, it is often impossible to determine the degree of trust which should be afforded to any initial problem description. Consequently, the determination of how abstractly to begin the design is one of the more difficult tasks for the software designer. Our methodology advocates an initial abstraction which is more general than appears to be indicated by the initial problem requirements. Although no reference could be found of any other software design methodology taking this approach, there

are a number of reasons for assuming that it is reasonable and logically warranted.

First, the proven inadequacy of initial requirement definitions and the evolving nature of requirements make it necessary to give the designer sufficient latitude to meet the client's *reasonable* requirements. This brings us to compare the additive with the limitative approach to design. The additive approach espouses the philosophy that the designer should adopt a meager initial design and add capability as the requirements become known. This is a very open-ended and undisciplined design technique. On the other hand, the limitative approach suggests that an initial *upper bound* be placed on the capabilities of the design and that excess capacity be deleted by design decisions as the requirements evolve. This is a much more bounded approach; therefore, it affords greater control over the design process since a finite limit on the design scope and capacity is predefined. As a simplified example of these two approaches, consider the design of an objective test scoring system. The additive approach might be to represent a test as a list (SEQ) of answers; while the limitative approach would begin more abstractly, perhaps with a test represented by a relation between question numbers and answers. In the former approach, the decision to allow questions with multiple or no correct answer will necessitate an addition to the formal representation of a test; however, the limitative abstraction could be restricted to accommodate this reasonable requirement. In practical use, a methodology must allow both approaches but should build where possible on the management advantages of the limitative technique. This suggests that an initial abstraction general enough to cover most reasonable client requirements could be used to constrain the problem and allow a limitative design technique.

Secondly, we must consider the client's comprehension of the system. Since our criteria demand that the client be involved throughout the design process, it is important that he understands and agrees to the initial design. The absence of detail which characterizes abstractions makes a very abstract design generally easier to explain and, hence, a better basis upon which to build the client's mental image of the design.

Thirdly, we should note that any formal design starting point will *hide* decisions by implicitly requiring them within the formalism. As a very simple example, an initial abstract design of a list (SEQ) of words hides the decisions that the words are to be consecutive and are numbered starting from one. A partial function from natural numbers to words as the initial abstraction

hides neither of those decisions. The partial function representation would have to be explicitly constrained to include these decisions. We insist that any significant decision must be explained in natural language before it is defined in the formalism of the design. Therefore, we must attempt to limit hidden decisions to those which are trivial or obvious by using very general initial abstractions.

Further, the flexibility and comprehensibility of a highly abstract design make it a good candidate for use as a starting point for other similar design projects. That is, software design cumulativity could, perhaps, be enhanced be using the same very abstract initial design for various similar design problems.

b. The Hierarchy.

The hierarchical nature of our design methodology is a result of a combination of two types of decomposition, horizontal and vertical[Goguen,80]. Horizontal decomposition partitions the initial requirements into modules in order to control the complexity and manageability of the design task. Vertical decomposition divides levels of detail in a module to separate and order the design decisions. For example, a report generator design might be horizontally decomposed into separate modules for data input, data base update, and report output. The data base update module could have one vertical level which records decisions about the required relationship among various data fields in the data base, with the next vertical level restricting the number of each type of record allowed in the data base. Horizontal decomposition strives to create totally independent modules while each vertical level is dependent upon the previous level for its basis.

b.1 Horizontal decomposition.

Horizontal decomposition reduces the design problem, where necessary, to a set of equivalent subproblems, each of which is conceptually less difficult. This type of decomposition might also be regarded as static decomposition because it is usually accomplished at the very onset of the design process and ideally is not changed until each subproblem is solved. At this point the static decomposition is undone by recomposition of the decomposed components. This recomposition is required to demonstrate that the various pieces of the design fit together in the manner contemplated when they were decomposed.

The problem of how to horizontally decompose a requirement influences the problem of how abstractly to define the decomposed subproblems. The approach to the static decomposition is usually a creative decision, based on experience and not directed by the methodology; hence, it needs positive management control. Decomposition is affected by the determination of which portions of the problem solution can be taken from previous design work. The implication is that the system manager should scan previous designs whose more abstract models may be valuable as starting points. If no suitable abstractions are discovered, the problem will need to be solved strictly by the unpredictable creative activity of design. However, if usable abstractions are found, the decision of how to decompose the original client's requirements may be directed by the availability of abstract models for modules included in a decomposition. The decisions on decomposition and activity of design (creative or reapplicative) must be made concurrently as a first step.

The problem of finding predesigned abstract models which fit possible components of the current requirements brings into focus an ancillary issue which should be noted. This is the creation and maintenance of an abstraction/implementation library. Such a library would greatly aid in efforts to increase the cumulativity of software development. The publication and standardization of design abstractions (especially very high level ones) would certainly assist the effort of creating an accumulation of abstract models. However, the marginal success gained by the several mathematical libraries of this type points out three requirements for the success of such a library: 1) it must be well structured and easily accessed (perhaps automated), 2) it must be well understood by its users, and 3) it must be in a common, standardized notation.

For large scale developments, the horizontal decomposition/recomposition process is especially important. Management of multi-man designs requires that horizontally decomposed modules must be (made to appear) independent of each other (decoupled). In many instances, it will not be possible to decompose the problem into decoupled modules of an appropriate size. In such cases, the effects of the coupling must be abstractly defined within each module. This technique may require verification at recomposition that each coupled module acts (or interacts) in the way it has been abstractly described in each of its coupled partners. That is, at recomposition the abstract representation(s) of a module within

other modules must be reconciled with the actual design for the module to produce a single consistent representation. Constant management attention is required on this very critical problem of component communication.

The timing of recomposition is another one of the creative decisions that must be developed through experience. For totally independent modules, recomposition is necessary only at the end of the design process. However, coupled modules may be recomposed when the advantage achieved by decomposition is outweighed by the necessity to share information or design decisions among modules. Also, intermediate recompositions may be useful to aid the manager's or client's perception of the system. Recomposition is done with the union of abstract designs (abstract machines) and their underlying theories. The new larger abstract design combines the attributes (ie. invariants, state components, generic parameter definitions, observations, and operations) of the decomposed abstract designs into the smallest possible complete set of attributes. That is, for example, identical state components appearing in two abstract designs which are recomposed will appear only once in the recomposed machine. (See "Shared Subtheories" in [Burstall,80] for a more detailed explanation of combining theories.)

b.2 Vertical decomposition.

Vertical decomposition denotes the different levels of design decisions and simplifications which are documented in the various steps (called advancements and refinements) of the specification process. Vertical decomposition might also be called dynamic decomposition because it is a process which continues throughout the design process. However, at each successive level of dynamic decomposition the designer is obligated to maintain the consistency between the new level and the previous level of the design. This constant check on the consistency of the successive levels of design removes any further requirement to recompose the various vertical levels.

In the specification of a software system, there are two basic areas of concern to the software designer. The first is the accurate, complete, and verifiable documentation of the system required by the client. The second is a design whose primitives can be easily and efficiently implemented. These two concerns are each sufficiently complex that the combination of them in a design methodology results in specifications which are often muddled and incomprehensible. Our methodology structures the design process so that

the separation of these concerns is possible. This separation eases the design process and facilitates the implementation of the resultant design. Advancement records and formalizes the design decisions while refinement simplifies the formal representation of the design.

Advancement.

When the designer takes decisions to move an abstract design closer to the client's required system, he restricts the different implementations (called the family of systems) specified by the abstract design. Taking these restricting decisions is the embodiment of the creative craft of limitative design. It is essential to record decisions as precisely as possible so the resulting system is accountable, verifiable, and maintainable. We advance the abstract design toward the required system each time a new level of decisions is taken and documented.

In many cases the order in which decisions are taken is defined or influenced by factors outside the control of the software designer -- like client requirement clarification, information input from external sources, etc. However, where possible the decisions documented by the hierarchy of advancement specifications should be directed by the rule that decisions about which the designer (or client) is most confident would be used to advance the specification first. This is an obvious technique for limiting the backtracking required when questionable decisions need to be rescinded. A second rule suggests that critical decisions should be identified, if not actually made, as early as possible[Parnas,72]. The order in which decisions are made is very important but cannot be legislated -- the experience and good judgment of the designer is an important factor in the proper ordering of decisions.

Refinement

The second area of concern to the designer has less to do with creative art of designing a system and more to do with insuring a design is not overly complex and is likely to be implementable on a concrete machine. The following picture illustrates the difference between advancement and refinement of a specification. (Advancement by decisions A and B restricts the legal machine states; the refinement creates a new but equivalent representation of the state space.)

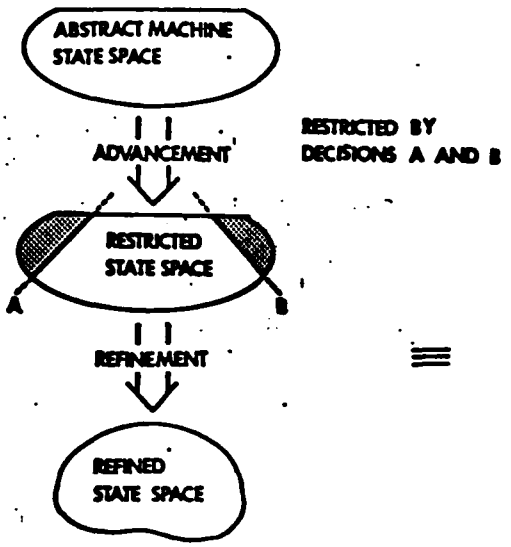


FIG. II-1 -- Advancement and Refinement.

The refinement process allows the specification of an abstract machine to be changed to an equivalent machine with a different representation[Jones,80]. In general, a refinement is used to make the machine representation slightly more concrete, although that is not always the case -- a simple regrouping of components for clarity is another refinement usage. More specific reasons for producing a refinement include the following:

1. After several advancements of an abstract machine the complexity and detail of the state space restrictions often becomes unwieldy and difficult to understand. At such a point, a refinement would be useful to create a machine where these restrictions are less cumbersome or are implicit in the structure of the machine. That is, a new state space equivalent to the remaining portion of the old state space is defined.

2. If new operations are difficult to define on the abstract machine as it currently exists (or old operations have grown unwieldy and difficult to understand because of past advancements) then is may be possible to refine the state space to facilitate the specification of the operations.

3. Implementation considerations might also prescribe the refinement of the abstract machine. These considerations are often efficiency considerations in the latter portions of the design process. For example, if an adequate replacement for the abstract machine exists which represents more closely the hardware or software features of the intended (concrete) machine, then this new representation is likely to be less expensive to implement in terms of design effort, system resource requirements, etc.

4. The initial decomposition of the design was done to structure the design effort and make it more comprehensible by separating the different concerns. At some point, the decomposed modules will need to be recomposed to form the total system. But before the different modules can be recomposed into a single abstract machine, the components held in common by these modules need to be refined into an identical representation in all modules. This will allow the common machine components to be represented by a single component of the recomposed machine state space.

From this discussion it can be noted that refinement is not a framework for documenting design (as is advancement) but is a tool for facilitating the readability and implementability of the design specification. The freedom to make refinement and advancement design steps in any order required by the problem is critical to the generality of our methodology. (See [Domolki,80] for a very brief analysis of the difference between the two theory relations we have called advancement and refinement.)

b.3 Summary.

The hierarchical nature of this design methodology is a result of the combination of the two types of decomposition, vertical and horizontal. The following diagram will help to demonstrate this point.

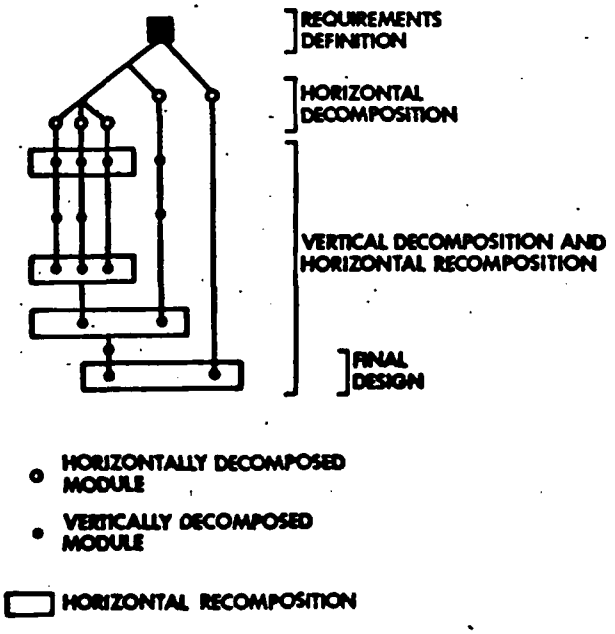


FIG. II-2 -- The Hierarchy

There may be many decompositions (vertical and horizontal) which are wrong for a particular problem; there certainly is not a single one which is right. Indeed, the best decomposition for a problem may well depend on such factors as client's understanding of the requirements, the software engineer's abilities, available abstractions, size of problem, etc. Horizontal decomposition may separate modules (abstract subproblems) which relate to different requirements set forth by the client. Or, it may separate out important parts of the state space for closer attention. Vertical decomposition can order decisions together by effect, by degree of difficulty, or by the probable stability of the decisions.

c. Formal Specification.

In order to unambiguously state the requirements and design of the desired system, we will use a formal specification language. Using a formal language allows us to contemplate design issues and perform design verification prior to writing any program code.

We selected an extensible set theoretical notation, sometimes referred to as "Z", for our formal specification notation. Although it is not executable, it does have the scope to describe theories at any level from the very abstract to a Pascal-like pseudo programming language. This notation is state based with parameterized theories. In it we have the capacity for both procedural and data abstraction.

For a more complete introduction to this notation see [Sufrin,82], [Morgan,82a], and [Morgan,82b]. A brief review of relevant aspects of the notation is presented in Appendix A.

d. Verification.

In our methodology, we must have the capability to formally and informally verify the products of the design process. The informal verification techniques include walkthroughs, audits, reviews, and program testing. Since these are commonly found in the literature (eg. [Yourdon,78]), we will not dwell on them except to explain how they are used in conjunction with the formal products of the design.

The informal techniques can be used to check the formal design in exactly the same manner as would be done in other methodologies to check informal designs. Of course, the participants in the review or walkthrough must be conversant in the formal notation and its semantics in order to verify

a formal design. If part of the design is brought into question by this informal verification, then formal checks could be instituted to clarify the parts in doubt. Such a two level verification approach focuses the mathematical exactness of formal verification at the critical points of the design, thereby reducing its expense.

The informal techniques could also be used whenever it is necessary to compare the formal documentation with other representations of the design. For example, it may be necessary to review the formal design relative to the provisional user's manual, to review the program code relative to the formal design and implementation plan, or to review the formal specification relative to its associated natural language explanation.

We can also use formal methods in several ways to verify a design. First, we can prove the correctness of successive vertical levels of the design. In the limitative design, the allowable conditions of the system (states of the abstract machine) must be reduced in number or held constant by each successive vertical level. That is, if S_1, S_2, S_3.... represent successive advancements of the state S, then state space (S_3) contains a subset of states in state space (S_2), which contains a subset of states in (S_1) (ie. ...(S_3) $\subseteq$ (S_2) $\subseteq$ (S_1) $\subseteq$ (S)). Advancement is the process of adding restrictions or predicates to the state (ie. $S_2 = S_1 \mid P$). Refinement is the process of defining a new state space representation which is homomorphic to the previous state space. The relationship between successive vertical decompositions, whether related by advancement or refinement, can be expressed by a retrieval function. (See [Jones,80] for a detailed discussion of retrieval functions or, similarly, see [Hoare,72; Sufrin,81a] for a discussion of abstraction functions.) In advancement, the retrieval function is the identity function restricted to a domain including only the new set of legal states. The retrieval function in refinement embodies the homomorphism relationship. With recomposition, the retrieval functions can be used to relate the relevant parts of the new state space to each of the component state spaces in the same manner as with advancement or refinement. In all cases, it is possible to demonstrate if a lower level design correctly models its predecessor.

The second method to formally verify a design is to state and prove theorems about the behavior of the abstract system and compare the proven behavior with the client's desire. This is the mode which would be most often used to answer client and management questions and to clarify potential problems noted by informal verification techniques. Because of the

relationship which exists between successive abstract levels, we choose the most appropriate design level to determine the system behavior and then are confident that this behavior is manifested in all lower level machines unless it is explicitly changed by a design decision. The greatest difficulty with formal verification of this type is that translating natural language questions to formal theorems may be a complex process requiring a deep and detailed understanding of the relationship between the formalism and reality.

Finally, we can prove the correctness of a program implementation relative to its formal design. A proof that the program correctly implements the requirements is the ultimate means of demonstrating reliability. Although substantial effort has gone into making program verification a practical technique, it is still not viable on large systems for two basic reasons. First, the cost and complexity of formal program verification make it economically infeasible on a large - scale[Berg.82; Pressman.82]. Secondly, program verification assumes the appropriateness of the formal specification; however, as we noted earlier, there is no formal way of validating that the specifications actually reflect the client's requirements. Therefore, a proven implementation may be a correct solution to the wrong problem.

Because of these two problems, we choose to deemphasize the role of formal program verification and reemphasize the other forms of verification. Additionally, we attempt to resolve the requirement validation problem by incremental requirement definition with client understanding and agreement. That is, if the initial design is so simple that it obviously satisfies the client's needs and if the successive levels of design can be proven (by retrieval functions) to accurately model that initial design, then only the design changes (advancements) need to be validated. Since these changes are done incrementally with constant client interaction and with properties of the design proven in the abstract machine, the likelihood of solving the right problem will be high. Therefore, we have shifted the emphasis of our methodology to the very earliest portion of the lifecycle -- the requirements definition and validation.

It is the intent of this methodology that formal verification will be separated from the design process so that management can control these two aspects independently. Indeed, the formal verification will be done by someone other than the designer (perhaps a mathematician) as an independent check of the design. However, to be of benefit to the designer and the manager, all verification must be accomplished in a timely manner and coordinated with the designer.

One final point needs to be made about verification within this methodology: management must control the degree of verification applied to individual modules of the design and to the system as a whole. That is, formal methods can be applied at different degrees of rigor[Jones,80], as can informal methods. The manager must balance the type and degree of these two forms of verification to best suit the project. We shall not attempt to define a standard or even a minimum level of verification required for all systems.

(The following are a few of the many possible references for the mathematics of formal verification: [Abrial,82], [Berg,82], [Dahl,72], [Dijkstra,76], [Gries,81], [Hoare,80], [Jones,80], [Richardson,82], [Sufrin,81c].)

II.3 Methodology Overview

This section will put the software engineering concepts discussed in the previous section into perspective by integrating them into a complete methodology. The first subsection explains the methodology in terms of the products of the design. As a cross-reference, the second subsection briefly outlines the methodology by defining the activities of each design participant during the software lifecycle.

We have divided the software lifecycle into six phases for the purposes of our methodology:

1. Requirements definition,
2. Problem decomposition,
3. Abstract design,
4. Detailed design,
5. Implementation and testing,
6. Operation and maintenance.

Within these phases we have included three baselines and five recommended product review points.

Baselines are part of a management concept which has been proven in large scale software development efforts -- it is the milestone/baseline/configuration management approach. Very briefly, this approach advocates setting incremental *targets* (milestones) in resource utilization and development completion which can be compared against actual results. As certain of these targets are reached, selected products of the

design (documentation, client agreements, design steps, code, etc.) are verified and *frozen* (baselined). They may then be changed only if approved through the configuration management system.

A configuration management system views the product in design as a configuration of components (modules). Before baselining the designer *owns* a component; after baselining it is *owned* by management, who then control further changes to that component. In this way, the manager will always be able to determine the current *approved* status (configuration) of the system. He can also authorize independent audits or additional reviews to recertify the configuration. (Further details on the topics of milestones, baselines, and configuration management can be found in most references on software development management; see especially [Boehm,81] or [Jensen,79].)

The advantages of this approach to the control of the management process, resource accountability, and product reliability are obvious. We are convinced that such a capability must be woven into any methodology if it is to be practically viable. Because of the incremental design technique of our methodology, milestones, baselines and a configuration management system can be integrated directly into the methodology to the degree desired by management. There are three points of control, or baselines, which are inherently part of our methodology -- the abstract design baseline, the detailed design baseline, and the implementation baseline.

a. The Products.

Requirement definition. The first phase requires the definition of user requirements in informal text. This documentation must correctly reflect the client's general needs; therefore, the emphasis will be on completeness rather than detail. A more detailed representation of the client's requirements, if available, would be useful to guide the designer in later phases but is not necessary at this point.

Problem decomposition. The first formal product of the lifecycle is a set of initial abstractions, one for each horizontally decomposed module of the system. With each initial abstraction will be an allocation of the requirements which must be satisfied by the following levels of design for that module. Other documents produced in this phase include the initial budget, the initial milestones, and the definition of standards.

At the end of this phase is the first review -- the Decomposition Review. It will assess the completeness of the requirement allocation, the suitability of the initial abstractions, and the degree of coupling among the modules. As a result of this review, the designers should understand their module's abstract starting point and how their design will interface into the system as a whole. The participants in this review must include all of the designers, as well as the manager and client.

Abstract design. The abstract design baseline occurs at the end of the design phase from the problem decomposition to the point where details specific to the client's problem (rather than a large class of problems) are decided. The design specifications that occur in the abstract design phase are those that might be found in an abstraction library. Unlike the other two, the timing of this baseline is not precisely defined for all modules but is based on the magnitude of the design problem.

The baselined products from the abstract design include the formal design specification, the verification documentation, the client documentation, and resource management documentation. One part of the client documentation which is baselined at this point is the provisional user's manual. This user's manual embodies the baselined abstract design and extrapolates that design to an implementation based on the designer's current understanding of the requirements. Naturally, this manual will not be accurate in every detail since most of the detailed decisions have not been taken; however, it is an excellent opportunity to compare the perception of the designer and the client, and force the client to begin to formulate the details of his requirements. (Boehm calls this technique "anticipatory documentation" and suggests that it can be highly effective in reducing lifecycle costs[Boehm,81].)

At the end of this phase but before the documents are baselined, the Abstract Design Review is held. This activity will review each formal abstract design for:

- an evaluation of the design approach
  - -- likely requirements are still reachable in the design
  - -- the design will lead to an implementable system
- consistency with the associated informal text
- compliance with standards
- an evaluation of the required formal verification.

Additionally, this review will establish the feasibility of the system defined by the provisional user's manual. This review is attended by designers, verifiers, the client, and the manager.

Detailed design. The design phase from the abstract design baseline until just prior to implementation is called the detailed design phase. It is in this phase that the experience and ability of the software engineer pays dividends. The main reason for this is that design (especially detailed) is rather like a funnel, with each design decision narrowing the working area. Consequently, the molecules (requirements) attempting to make their way through the funnel begin to interact and interfere with each other. Unless fruitful compromises are discovered among these molecules, some will be thwarted as others take precedence. Since the specific requirements of a system especially the non-functional requirements and implementation factors, are never mutually exclusive, the designer's difficulties are greatly compounded by this need to seek compromises and trade-offs among the various factors he must consider simultaneously. There is a tool which can be used in conjunction with this methodology which allows the designer to evaluate the design compromises and assumptions he makes. This tool is called prototyping.

Prototyping in the context of this design methodology is the creation of a model which exhibits the salient features of the system being examined. The model may be mental, as produced by the provisional user's manual developed in the abstract design, or it may be physical, like the creation of a display algorithm which is executable on the target machine and from which display characteristics and efficiencies can be checked. In general, we will speak of prototypes as models which are created in parallel with the formal design process and usually on a smaller scale than a real implementation. (This is the second technique, which along with anticipatory documentation, will aid in the reduction of software costs according to [Boehm,81]. Boehm calls extra products developed in parallel with the early design process "scaffolding.")

As noted above, one of the advantages of prototyping is the insight provided to the designer concerning the appropriateness of the compromises he has developed among the various requirements of the system. A second advantage accrues to the client -- the prototype gives him a model of the designer's system which he can compare against the system he desires.

One form of prototype which can be used in this way is the *"What if...?* question. The client presents a situation and the designer determines exactly how the abstract machine reacts to that situation by formally rephrasing the question and defining the result from the system specification. The final advantage is that prototype experiments can be used to answer specific questions, such as: "Can our design be limited by this simplification and still fulfill the client's needs?" Since ours is a generally limitative methodology, the answers to such questions are very important.

It is conceptually possible for the detailed design to be produced to such a level of detail that the implementation process becomes a trivial rewriting through the use of standard data structures and notational transformations. However, in most instances, the verifiability and ease of implementation gained by such a minute level of detail is vastly overwhelmed by the cost of undertaking such an intense design effort. The design manager must decide for each individual design problem what balance of cost and formal verifiability is most suitable. In other words, he must determine when the marginal utility of a further level of detail is too low to warrant proceeding further in the formal design process. In general, this will mean taking a few relatively large design steps with another large step to an implementation in a structured language (even if the ultimate implementation is not in a structured language.) At this point the gap between the formal (verifiable) design and the implementation can be bridged with the implementation notes or by selection of previously verified algorithms from an algorithm library. The implementation notes will include the appropriate invariants under which the algorithms not yet totally specified must function and prototype experiments demonstrating which algorithm(s) might best satisfy the client's requirements in a specific module.

It is during this phase that the test plans are developed for each module and the system as a whole. The overall approach to test planning depends on the balance the manager has struck between formal and informal verification; however, informal testing against non-functional requirements (eg. implementation speed, useability, etc.), integration testing, and testing of abstractly specified or unspecified algorithms will always be necessary.

A test plan must include a definition of the purpose of the test, the method of testing (including whether formal or informal), the inputs required, the outputs expected, and the criteria for success. The outputs expected should reflect the coincidence of the client's expectations and the behavior

formally predicted from the specification. It is important that the test planning be accomplished in conjunction with the design because no one will have a better feel for potential problem areas than the designer and his independent auditor, the verifier.

The baselined products of the detailed design phase include formal specification documentation containing an abstract design to display the client's requirements, a test plan, implementation notes to guide the implementation of those portions of the system which are not immediate from the formal specifications, and the latest verification, client and resource management documentation.

The Detailed Design Review assesses the detailed design with regard to:

- the design approach
  -- the order and structure of design decisions is appropriate
  -- likely requirements changes, non-functional requirements, and fine tuning can be accommodated in the design
- consistency with the associated informal text
- compliance with standards
- compatibility with the provisional user's manual
- the appropriateness of the bottom level of design
  -- state structure and operations are appropriate to the hardware and programming language
  -- step size to the implementation is appropriate for all algorithms

This review must also evaluate the completeness of the test planning/scheduling and the implementation notes to be used in the next phase. It is important that all design participants, including the implementers and the client, are involved in this review.

Implementation and testing. The implementation baseline follows the coding and acceptance verification of the system. The products frozen here include all previous design documentation, resource usage documentation, verification results, final system documentation (eg. user's manual, operation manual, maintenance manual, etc.), and the system code.

During this phase the products are reviewed twice to establish their correctness. The first review, called the Software Verification Review, establishes that the software is ready for testing. The major task of this review is to insure that the software models the formal specification. For

small important modules this can be done formally; however, in general it will be done using informal review (walkthrough) techniques. The designers, verifiers, and implementers are the key participants in the Software Verification Review.

The second review of this phase occurs just prior to the implemention baseline. The System Review will establish that the development products are ready for release to the client. In this assessment the results of the system tests are analyzed and the final user documentation is matched against the software and its formal design. The client serves as the approval authority in this review. After successful completion of this review, the system and its documentation is handed over to the client and maintenance personnel.

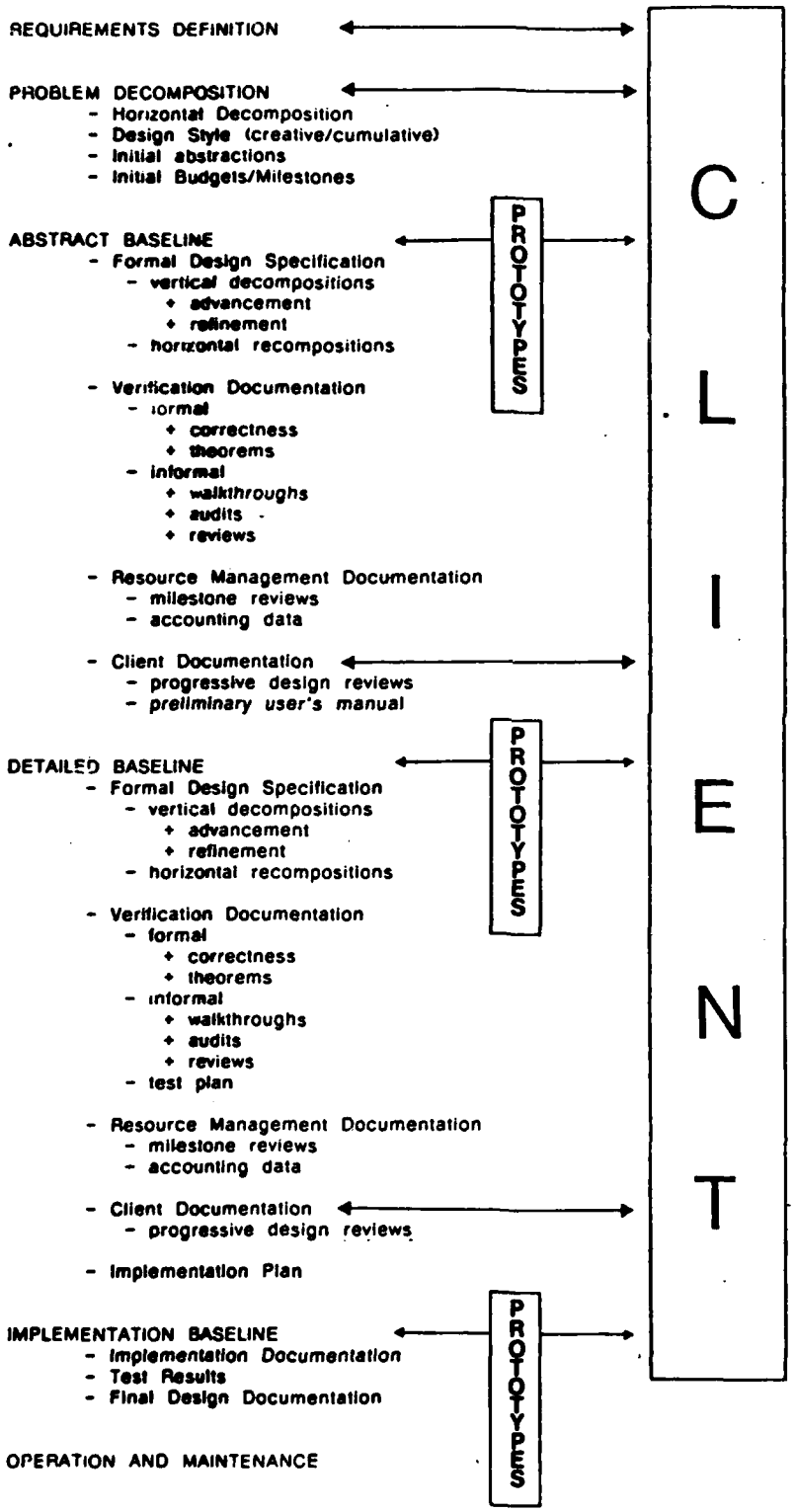


FIG. II-3 -- The Lifecycle and the Development Products.

b. The Activities.

This outline briefly defines the activity of the design participants in each phase of the software lifecycle. Those activities which are preceded by an asterisk relate to the formal aspects of the methodology and are presented in slightly more detail.

Requirements Definition

Manager

1. Collect requirements data from the client or systems analyst.

Problem Decomposition

Manager

* 1. Search the abstraction library. Find previous designs which might be useful as a component design for the current problem.

* 2. Decompose the requirements. Identify and separate modules which must be creatively designed from those which can use previous designs.

* 3. Define initial abstractions or reapplication starting points and module requirements for each module.

4. Set standards:

* a. documentation. Include the standard for the formal specification and verification documentation.

* b. vertical decomposition granularity for each module. Define the limits of the vertical step size which are appropriate for the problem.

* c. level of final abstract for each module. Define the approximate level of design to be specified formally.

d. configuration management.

* e. type/degree of verification for each module. Determine if correctness verification is required and at what level of rigor. Define fixed review (walkthrough, audit) points and criteria -- including the formal specification reviews.

5. Develop initial budget.
    a. allocate resources to modules for design.
    b. staff project/modules.
6. Set initial milestones.
7. Chair Decomposition Review.

Client

1. Agree with initial design.

Designers

1. Understand standards.
* 2. Understand abstract starting point for their module(s).

Abstract Design

Designers (for each module)

* 1. Vertically decompose and specify module through its initial abstract steps. Provide the advancement and refinement steps for the module at the desired granularity. Document each level and emphasize the design decisions taken, their rationale, and alternatives considered, and justify the order of decisions using the appropriate documentation convention.
2. Develop provisional user's manual for module.

Client

1. Agree with design decisions.
2. Agree with provisional user's manual(s).
3. Ask "What if..." questions.

Verifiers (for each module)

* 1. Do correctness verification of each level. If required by the management standards, insure the correctness of the module specification by continuous formal verification. This must be done to the degree of rigor indicated by the manager. Results of verifications are reported quickly to designer. Significant problems are reported to the manager.

* 2. Prove theorems set by manager or designer. As a result of informal verification or questions about the behavior of the system, properties of the system may need to be formally demonstrated.

* 3. Fill in design as required. The verifier may be charged with finishing details of the formal specification left implicit by the designer. For example, the verifier can do trivial promotion of operations, add explicit preconditions to operations as required by advancement, or do substitutions and simplifications required by refinement since these do not involve design decisions.

Manager

1. Monitor:
    a. designer/client interaction.
    b. intermediate resource usage.

* c. intermediate design progress. To monitor the design he can read the natural language or formal documentation of the design decisions, set up intermediate reviews or walkthroughs of the design (specification), set theorems or ask questions about the behavior of the system, and review the formal verification documentation.

    d. standards
    e. milestones

2. Direct integration of provisional user's manual.
3. Authorize and budget the production of prototypes.

* 4. Direct required horizontal recomposition. For those modules which must be recomposed prior to the completion of their design, the formal recomposition must be initiated and resources reallocated to reflect a new design structure.

Implementors

1. Produce prototypes as required.

Abstract Baseline

Manager

1. Review:
    a. resource usage.
* b. design correctness. Chair the Abstract Design Review.
2. Redefine:
    a. milestones.
    b. resource allocation and budgets.
3. Put appropriate documents under configuration control.

Detailed Design

Designers (for each module)

* 1. Vertically decompose and specify module through its final abstract stage. Provide the advancement and refinement steps for the module at the desired granularity. Document each level and emphasize the design decisions taken, their rationale, and alternatives considered, and justify the order of decisions using the appropriate documentation convention.

* 2. Develop test plan. The test plan will concentrate effort on areas left abstract in the formal design and on areas of complexity in the specifications. Test plans can indicate behavior predicted by the formal design to various test cases.

* 3. Develop implementation plan. The implementation plan will provide insight to the implementors about the implementation of abstractions remaining in the formal design, possible initializations, and potential techniques for accommodating non-functional requirements.

Client

1. Agree with design decisions.
2. Ask "What if..." questions.

Verifiers (for each module)

* 1. Do correctness verification of each level. If required by the management standards, insure the correctness of the module specification

by continuous formal verification. This must be done to the degree of rigor indicated by the manager. Results of verifications are reported quickly to designer. Significant problems are reported to the manager.

* 2. Prove theorems set by manager or designer. As a result of informal verification or questions about the behavior of the system, properties of the system may need to be formally demonstrated.

* 3. Fill in design as required. The verifier may be charged with finishing details of the formal specification left implicit by the designer. For example, the verifier can do trivial promotion of operations, add explicit preconditions to operations as required by advancement, or do substitutions and simplifications required by refinement since these do not involve design decisions.

Manager

1. Monitor:

   a. designer/client interaction.

   b. intermediate resource usage.

* c. intermediate design progress. To monitor the design he can read the natural language or formal documentation of the design decisions, set up intermediate reviews or walkthroughs of the design (specification), set theorems or ask questions about the behavior of the system, and review the formal verification documentation.

   d. standards.

   e. milestones.

2. Direct integration of module test plans and develop the test schedule.

3. Direct integration of implementation plans.

4. Authorize and budget the production of prototypes.

* 5. Direct required horizontal recomposition. The modules must be recomposed prior to the completion of the design. The formal recomposition must be initiated at the appropriate time.

Implementors

1. Produce prototypes as required.

Detailed Baseline

Manager

1. Review:
    a. resource usage.
    b. design correctness. Chair the Detailed Design Review.
2. Redefine:
    a. milestones.
    b. resource allocation and budgets (allocate implementation resources).
3. Put appropriate documents under configuration control.

Implementation and Testing

Implementors (for each module)

1. Develop code according to the formal specifications and the implementation plan.

Verifiers (for each module)

1. Test according to test plan.
2. Assess implementation anomalies against the behavior documented in the formal design.

All

1. Prepare final system documentation.

Manager

1. Monitor:
    a. intermediate resource usage.
    b. intermediate implementation progress/test results.
    c. standards.
    d. milestones.
2. Direct recomposition of implementation modules.
3. Authorize and budget production of prototypes.
4. Chair the Software Verification Review.

Implementation Baseline

Manager

1. Review:
    a. resource usage.
    b. implementation correctness. Chair the System Review.
2. Put appropriate documents under configuration control.

Operation and Maintenance

Manager

1. Manage the transition to the new system.
2. Direct fine tuning and initial modification efforts.
3. Evaluate the project:
    a. accuracy of resource estimates.
* b. value of initial abstractions.
    c. personnel critique.
    d. general lessons learned.
* 4. Direct updating of the abstraction library.

Maintainer

* 1. Refer problems to the formal specifications. The behavior of the formal design will indicate how the implementation should function and will determine if the problem is one of implementation or design.

2. Cost of design adaptation is estimated based on resource data attached to specification design levels.

3. Maintain the specifications.

II.4 A Small Example.

A further explanation of the proposed methodology is woven into the very small example development which follows. This development is meant to be a gentle introduction to the style and structure of the design and specification techniques only. In order to keep it small, many of the decisions have been made for simplicity rather than for a practical reason. The reader should keep in mind that the methodology proposed in this thesis was designed for large scale system developments and, hence, it may seem overbearing on such a small example. For purposes of introduction, the horizontal and vertical decomposition done here will generate module granularity which is much smaller than is normally practical.

A management tool which is commonly used in the documentation of a design process is a standard form or documentation template. The exact format of the template is not nearly as significant as the general discipline which is imposed by its use and the management control which it allows. The template is a checklist to remind the designer of his obligations in producing documentation and in maintenance of the management defined standards for the design. For the manager, it differentiates the various design steps to allow incremental control of the formalism, standards, and resource utilization.

Of course, any template usage is complete only if natural language explanatory text is presented to interpret the mathematics. This interpretation should be geared to the future audience -- the design manager, the software maintainer, the implementer, and other designers. Although this text is not written specifically for the client, it is normal to use these textual explanations as the basis for the user documentation.

A blank sample of the documentation template used in this thesis is presented on the next page. The sections of the template are explained as they are used in the following simple design example. Each use of the template represents a single level of vertical decomposition.

NAME

Management Data:

Author:

Date Started: Date Completed:

Resources Expended:

Project Name:

Verification Documentation Accomplished:

Client Documentation Accomplished:

Basis:

Library:

Prior:

Forward:

Comments:

- -

Requirements/Design Decisions:

Auxiliary Definitions:

State (Component) Definition:

Observation Definitions:

Operation Definitions:

FIG. II-4 -- Design Template

Requirements Definition

Design and implement a document display system with line editing features for a personal computer. The system will allow textual scrolling, plus simple user commands for the creation, deletion, and insertion of lines of text.

Decomposition

This problem decomposes easily into two main modules -- the document display and the document editor. An appropriate abstraction of the document editor currently exists in our expanded set theory; therefore, only the document display module needs to be creatively developed. The lower level concerns such as hardware, environment, system, and user interfaces exist as components of almost every system; therefore, we refer to them as standard components. These standard components need not be explicitly defined in the decomposition step and will commonly be recomposed in the final levels of design. The following figure represents the decomposition for this simple problem.

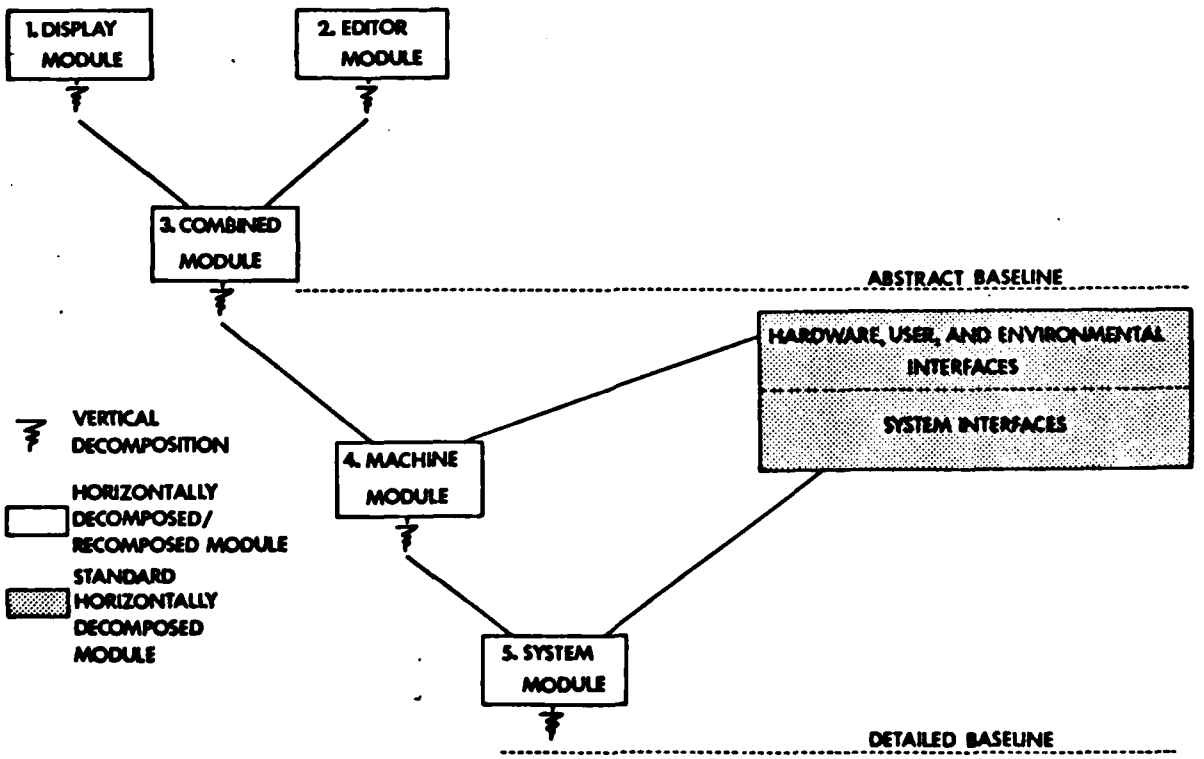


FIG. II-5 -- Problem Decomposition.

ABSTRACT DESIGN

1. The Display.

This is the first of our horizontally decomposed problem components.

DISPLAY

(DISPLAY is the name of this vertically decomposed module.)

Basis:

Library: Schema combinators

We must look to our specification library to find the meaning of any schema combinators which are used. The remainder of the operators are from the *standard* extended set notation ("Z"); these operators can be referenced in [Abrial.80b], [Abrial.82], and [Morgan.82a], with a brief overview in Appendix A.

This section will facilitate maintenance of the design by creating a *linked list* of the design levels.

Comments:

This specification defines a very simple, one document file, any part of which can be displayed to the user. The user can control the parts of the document that he sees.

This section explains the features included in this level of design and relates this level to the previous one.

Auxiliary Definitions:

Here we define any required background mathematics or *environmental* attributes such as generic parameters.

a. `lines` is ABSTRACT

At this abstract level it is not important what the exact nature of a document line will be. It may be a series of blanks and dots, a succession of different colors or hues, or, more simply, a sequence of alphanumeric characters. Until we give a specific definition to `lines` it will be a *generic variable*.

State Definition:

We have defined in this section the schema representing the abstract machine state space. The definition given is based on the background and environmental attributes defined above.

DISPLAY

doc	:	$\mathbb{N}_1 \nrightarrow$ lines
displayed	:	$\mathbb{P}(\mathbb{N}_1)$

The document is a partial function from line numbers to the lines they represent. The lines selected by the the user are designated in displayed.

Observations:

This section gives us the opportunity to create windows onto the machine so that we can *look* at (but not affect) its various interesting parts at any time while it is running.

ΦDISPLAY

DISPLAY

DISPLAY'

ΘDISPLAY' = ΘDISPLAY

This is the generalized *null operation* on which observations are based (ie. the state is not affected by the observation.) Henceforth, this standard Φ schema will not be explicitly defined.

a.

APPEARANCE

ΦDISPLAY

picture' : $\mathbb{N}_1 \rightarrow\!\!\!\!\!\!\rightarrow$ lines

---

picture' = doc ↾ displayed

The user will see that portion of the document which he has designated to be displayed. Note that this observation could also have been made a derived component of our state space and would have been *automatically* updated with each state change. Choosing to make the user's picture an observation forces us to *request* a picture update each time the state changes -- and that is more typical of what actually happens in a display system implementation.

Operations:

Here we represent the transition functions (events) which cause our abstract machine to move between points in the state space.

In this module of the specification, we are concerned with what the user sees of the document, not in changing the document itself. Consequently, we do not want the state component doc to be affected by the following operations. Therefore, we adhere to the rule that operations should be defined on the lowest reasonable state component (in the following case, displayed) and should not be promoted to the full state until necessary to complete the design.

ΔDISPLAYED

displayed,

displayed',

lineset : $\mathbb{P}(\mathbb{N}_1)$

status' : STATUS

---

status' = SUCCESS

This is the *delta* (or change) operation which provides a shorthand notation for our operation definitions.

The status output will allow us to provide for any error conditions which may be necessary. Advancement of the design may require us to restrict the domain of our machine operations; however, our machine must, in the end, be robust enough to allow all operations to be *attempted* from any element of the machine state space. Hence, error results are a required aspect of our design. The set of status conditions, STATUS, is a standard set which cannot be fully specified until the design is complete. SUCCESS is an abstract element of STATUS which signifies that the operation was attempted from a state which satisfied the operation's preconditions and, therefore, that the operation succeeded.

a.

ADDDISPLAY

ΔDISPLAYED

---

displayed' = displayed ∪ lineset

This allows more lines to be designated as visible to the user.

b.

DELETEDISPLAY

ΔDISPLAYED

---

displayed' = displayed - lineset

This allows lines to be removed from the set of lines displayed to the user.

DISPLAY1

(This level of vertical decomposition is an advancement of the previous design level.)

Basis:

Library: Schema combinators

Prior: DISPLAY

The abstract machine developed here will be based on the abstract machine developed in DISPLAY.

Comments:

We now make the specification less abstract (reduce the *family of systems* specified) by advancing the theory. We will require that all legal documents consist of lines whose line numbers are consecutive.

Requirements / Design Decisions:

This section informally enumerates the design decisions which will be formally reflected in this design level. All significant changes to the abstract machine and the rationale for these changes should be stated informally here.

[[1]] The line numbers in a document will be consecutive, beginning at one.

State Definition:

[[1]] DISPLAY1 ≙ DISPLAY | doc ∈ SEQ[lines]

The type of the state component doc is changed from the general partial function to the sequence partial function, giving the consecutive line numbers that we desire.

Notice that the formalization of the design decision is annotated by the number corresponding to the informal description of that design decision.

The observation and operation defined in DISPLAY are implicitly updated by this change to the abstract machine. In general, the precondition required by an advancement such as this can be determined by *forcing* the new postconditions back through the operations and observations.

DISPLAY2

Basis:

Prior: DISPLAY1

Comments:

We can now make more precise the arrangement of the document lines as displayed to the user. The picture a user would most naturally desire to see is one which displays a consecutive area of the document.

Requirements / Design Decisions:

[[1]] The area to be displayed will be a consecutive area of the document.

State Definition:

DISPLAY2

DISPLAY1

[[1]] $(\exists m,n : \mathbb{N}_1)$ (displayed = m..n)

This slightly less general state requires that the lines the user will see are consecutive.

DISPLAY2A

(This vertical decomposition is a refinement of the previous design.)

Basis:

Prior: DISPLAY2

Comments:

Even at this very abstract level it is possible to introduce a refinement which will aid the understanding and perhaps increase the efficiency of an eventual implementation. This refinement introduces the end points for the set of consecutive lines designated by `displayed`.

State Definition:

DISPLAY2A

DISPLAY2

start	:	$\mathbb{N}$
stop	:	$\mathbb{N}$

`displayed = start..stop`

This definition of `start` and `stop` allows us to either show or forget the potentially space-inefficient component `displayed` -- that is, `displayed` has become a derived component. Derived components can be used or ignored as required by the implementation.

2. The Editor.

This is the beginning of the second horizontally decomposed component of the system. the editor.

EDITOR

Basis:

Library: SEQ / Schema combinators

Prior: DISPLAY (Auxiliary Definitions)

Only the auxiliary definitions of DISPLAY (in this case the fact that `lines` is abstract) are required as the basis of this level; the DISPLAY abstract machine is not used here.

Comments:

This abstract specification represents the simplest possible editor. It is a portion of the operation set from type SEQ. For ease of reference. the *sequence editor* is extracted from the library and reproduced here.

State Definition:

EDITOR

doc	:	SEQ[lines]

The document is simply a list of lines.

Operations:

ΔEDITOR

EDITOR

EDITOR'

spot : $\mathbb{N}_1$

status' : STATUS

---

spot $\in$ 1..length(doc)

status' = SUCCESS

The delta operation requires that the line number selected for editing correspond to an existing line in the document.

BADSPOT?

ΦEDITOR

spot : $\mathbb{N}_1$

status' : STATUS

---

spot $\notin$ 1..length(doc)

status' = ERROR1

where ERROR1 = "The designated line does not exist in the file and cannot be edited."

This is the first instance in our design where an error condition is required. Error conditions will direct that a status other than SUCCESS be output. As a rule, we will define the appropriate user error messages as the error conditions are specified. This is done because the designer has a much better perspective of the error cause than the implementer and

will generate more meaningful error messages than would be possible at implementation. Error conditions will normally not result in a change in the machine state but are simply observations of the machine *(null* state transitions).

The error condition presented here results from an illegal line number being designated for the editing operation.

a.

INSERT

ΔEDITOR
line : lines

---

doc' = insert(doc,spot,line)

This operation adds a new line at the selected spot and moves all following lines ahead one position in the document.

The complete (total function) INSERT operation requires the addition of the error condition BADSPOT?. Therefore, the complete operation would be:

INSERT ≙ INSERT ∨ BADSPOT?

b.

DELETE

ΔEDITOR

---

doc' = delete(doc,spot)

DELETE removes the indicated line from the document and *closes* the hole caused by the deletion.

This operation also requires the addition of the error condition.

DELETE ≙ DELETE ∨ BADSPOT?

c.

ADD

EDITOR
EDITOR'
line : lines

status' : STATUS

---

doc' = doc * ⟨line⟩

status' = SUCCESS

This operation adds a new line at the end of the document.

3. The Combined Module.

This level rejoins the two horizontally decomposed modules just designed.

COMBINED

Basis:

Prior: DISPLAY2A / EDITOR

Comments:

This specification recomposes the display and editor modules and completes the abstract design of the system. This module will be the abstract design baseline and, therefore, will consolidate the design as it currently exists. That means that all observations and operations must be explicitly promoted (if required), advanced, and refined to the current abstract machine level so that they can be verified.

Requirements / Design Decisions:

[[1]] The length of the document must be limited.

Auxiliary Definitions:

a. maxdoclength $\in \mathbb{N}_1$

The document length has an upper bound which is constant but has not yet been determined.

State Definition:

COMBINED

DISPLAY2A

EDITOR

[[1]] length(doc) $\leq$ maxdoclength

which we can expand to:

COMBINED

doc : SEQ[lines]

start : $\mathbb{N}$

stop : $\mathbb{N}$

%%% derived

displayed : $\mathbb{P}(\mathbb{N}_1)$

[[1]] length(doc) < maxdoclength

displayed = start..stop

Note that the document from the editor and the document defined in the display module have been merged in the combined state space. This can be done quite simply because the docs were both of the same type (ie. SEQ[lines]) with no contradictory environmental attributes (ie. lines was defined consistently in the two cases). In addition to merging the state components, this new schema includes the conjoined axioms of the editor and the display modules.

Observations:

a. The appearance observation is now:

APPEARANCE

ΦCOMBINED

picture' : $\mathbb{N} \nrightarrow$ lines

---

picture' = doc $\upharpoonright$ (start..stop)

This observation has been *promoted* to the new combined state from the DISPLAY state. Additionally, the derived component, displayed, has been replaced by its meaning, start..stop.

Operations:

All previous operations are consolidated and produced as functions of the appropriate state component:

ΔSTART-STOP

start,
stop,
start',
stop' : $\mathbb{N}$
lineset' : $\mathbb{P}(\mathbb{N}_1)$

status' : STATUS

---

status' = SUCCESS

In the first two operations, we need only substitute for the derived component and use this new delta function which is equivalent to the previous ΔDISPLAYED.

a.

ADDDISPLAY

ΔSTART-STOP

(start'..stop') = (start..stop) ∪ lineset

This is the updated version of the operation from the DISPLAY module. Notice that the decision to display a consecutive area of the document constrains the new set of lines which can be added by this operation.

b.

DELETEDISPLAY

ΔSTART-STOP

(start'..stop') = (start..stop) - lineset

This is the updated version of the operation from the DISPLAY module. The set of lines removed from the display must result in a consecutive display area.

c.

INSERT

$INSERT_{EDITOR}$

length(doc') ≤ maxdoclength

The previous INSERT definition had to be given the appropriate preconditions to adhere to all restrictions set in subsequent advancements of the machines. In this case the precondition is designed to make the state transition preserve the document length limit.

DOCTOOLONG?

ΦEDITOR

status' : STATUS

length(doc) = maxdoclength

status' = ERROR2

where ERROR2 ≙ "The document is already at its maximum length."

This error condition specifies that the document cannot be extended beyond the maximum length which has been set for it. Notice that we have not had reason to promote the editing operations past the EDITOR abstract machine, so this error specification is also presented at the EDITOR machine level.

With this error condition, the insert operation becomes:

INSERT ≙ INSERT ∨ DOCTOOLONG?

d. $DELETE \triangleq DELETE_{EDITOR}$

The previously defined DELETE operation is unaffected by the advancements.

e.

ADD

ADD_{EDITOR}

length(doc) < maxdoclength

This operation is updated to maintain the document length limitation. The total operation, including the error condition would be:

$ADD \triangleq ADD \vee DOCTOOLONG?$

DETAILED DESIGN

COMBINED1

Basis:

Prior: COMBINED

Comments:

Up to this point the design has been very abstract with no decisions made which were specific to this project. One of the ways the design was kept abstract was by making it generic on `lines`. We will now begin the detailed design for our specific problem by defining this generic parameter.

Requirements / Design Decisions:

[[1]] Each line in our document will be composed of a sequence of characters.

[[2]] There must be a limit on the maximum length of a line.

[[3]] We must be able to construct new lines for insertion into the document.

Auxiliary Definitions:

a. `chars is ABSTRACT`

We will not yet define what a character will be in this system.

b. `lines ≡ SEQ[chars]` [[1]]

Every line in the document is a list of characters.

c. `maxlinelength` $\in \mathbb{N}_1$

There is a constant which represents the maximum length of any line.

State Definition:

COMBINED1

COMBINED		
newline	:	lines

[[2]] (∀x : 1..length(doc))
(length(doc(x)) ≤ maxlinelength)

[[2]] length(newline) ≤ maxlinelength

This state demonstrates the first instance where the design progresses in a constructive rather than a limitative fashion. We have found it necessary to add a new component, newline, to the abstract machine. This new component represents a new line which we can add to the document. All lines, including the new line, must adhere to the line length restriction.

Operations:

a. [[3]]

ADD-CHAR

newline,		
newline'	:	SEQ[chars]
nextchar	:	chars
status'	:	STATUS

length(newline) < maxlinelength
newline' = newline * ⟨nextchar⟩

status' = SUCCESS

This operation allows us to add characters to the end of the new line. In this way, we can create any sequence of allowable characters as a line to add to the document. In order to add this newly created line to the document it must conform to the line length limitation.

LINETOOLONG?

newline,
newline' : SEQ[chars]
nextchar : char

status' : STATUS

---

length(newline) = maxlinelength

status' = ERROR3

where ERROR3 ≙ "The new line is already at its maximum length."

The error condition LINETOOLONG? is required for possible attempts to add an additional character to a new line which is already the maximum length.

The total operation becomes:

ADD-CHAR ≙ ADD-CHAR ∨ LINETOOLONG?

4. The Machine Module.

This level will join standardly decomposed aspects of the design (such as hardware, environment, and user components) to the current design.

MACHINE

Basis:

Library: Schema combinators

Prior: COMBINED1

Comments:

This specification will add the machine and environment interfaces to the design. The decisions taken here are not complete or realistic but serve to reduce the complexity of the example.

Requirements / Design Decisions:

[[1]] The character set of the target machine, LSI-11, will be used.

[[2]] The display surface is 24 lines of 80 characters (as required by the target VDU.)

[[3]] In order to avoid panning, the maximum line width should be the same as the screen width.

[[4]] The maximum document length is 50 lines due to the target machine memory limitations. (We will assume no disk storage is available.)

[[5]] All modifications of the document will be done on line 12 of the screen.

[[6]] The user's display will show as many of the user selected lines as possible while ensuring that one of the selected lines is displayed at the *action line* of the screen (line 12).

Auxiliary Definitions:

```
a.      [[1]]
char  ≙  {a,b,c,d,e,f,g,h,i,j,k,l,m,
          n,o,p,q,r,s,t,u,v,w,x,y,z,
          A,B,C,D,E,F,G,H,I,J,K,L,M,
          N,O,P,Q,R,S,T,U,V,W,X,Y,Z,
          1,2,3,4,5,6,7,8,9,0,
          !,",£,$,%,&,7,(,),_,=,-,
          ~,↑,|,\,',@,{,[,+,;,*,:,
          },],<,,,>,.,?,/ }
```

The allowable character set has been prescribed by the target machine.

```
b.      maxlinelength ≙ 80      [[2,3]]
        screenlength ≙ 24       [[2]]
```

The display surface is fixed by the target machine.

```
c.      maxdoclength ≙ 50       [[4]]
```

The constant maximum document length has been set.

```
d.      actionline ≙ 12
```

The physical screen line in which all document changes will be done has been set as twelve.

State Definition:

MACHINE

COMBINED1

cursorspot : $\mathbb{N}$

[[6]] start = 1

stop = length(doc)

[[5 & 6]] (doc ≠ ⟨⟩) ⇒

cursorspot ∈ (1..length(doc))

(doc = ⟨⟩) ⇒ (cursorspot = 0)

This new state space "recomposes" the previous state with the required *standard* components of the machine or machine environment. In this simple case, only the notion of a cursor position on the document is required; in more complex systems, such components as printer files, screen representations, main or secondary storage representations, keyboards, etc. may be required.

For this design, cursorspot will always point to the document line which appears at line 12 of the screen.

Observations:

a.

SCREENOFFSET

ΦMACHINE

picture : $\mathbb{N} \rightarrow\!\!\!\!\!\!\rightarrow$ lines

screen' : $\mathbb{N} \rightarrow\!\!\!\!\!\!\rightarrow$ lines

---

screen' = picture ∘

(shift(actionline-cursorspot) ⊳ (1..screenlength))

---

b. SCREENAPPEARANCE ≙ $\text{APPEARANCE}_{\text{MACHINE}}$; SCREENOFFSET

This new observation builds on the previous appearance observation. Now, given the current machine state, we can determine what line of the document (if any) is visible on each line of the VDU. This is done by restricting the previously defined `picture` to those document lines which will fit on the 24 lines of the screen, based on the knowledge that the `cursorspot` line always appears at line 12 (`actionline`) of the screen.

The observation APPEARANCE referenced in SCREENAPPEARANCE has been trivially promoted the the current state as indicated by the MACHINE subscript.

Operations:

a.

CURSORMOVE

```
cursorspot,
cursorspot' :        ℕ
amount      :        INT

status'     :        STATUS
```

```
(cursorspot + amount) ∈ (1..length(doc))
cursorspot' = cursorspot + amount

status' = SUCCESS
```

With this operation we can move the cursor position in the document in either direction as long as it stays on a line which has been selected for viewing [[6]]. This is the specification of the standard cursor movement operation adapted for our specific use.

OUTOFVIEW?

cursorspot,
cursorspot' : $\mathbb{N}$
amount : $\mathbb{INT}$

status' : STATUS

(cursorspot + amount) $\notin$ (1..length(doc))
cursorspot' = cursorspot

status' = ERROR4

where ERROR4 $\triangleq$ "The movement cannot be done because it would not leave a line visible at the actionspot."

This is the error condition corresponding to an attempt to move the cursor position such that it would not point to a *visible* document line.

The total operation would be:

CURSORMOVE $\triangleq$ CURSORMOVE $\vee$ OUTOFVIEW?

MACHINE1

Basis:

Library: Schema combinators

Prior: MACHINE

Comments:

This level of specification will design the actual user commands required in the system. These commands will be based upon previously designed state transition operations. This specification represents the detailed design baseline and will be the final groundwork for the total system design.

Requirements / Design Decisions:

[[1]] The following user commands will be allowed:

a. insert a line

b. add a line

c. delete a line

d. scroll up one line

e. scroll down one line

f. create a line with which to update the document

State Definition:

Unchanged from MACHINE.

Observations:

Unchanged from MACHINE.

Operations:

[[1]] All user required commands must be promoted to the MACHINE state space and must be made total functions with the addition of the necessary error conditions.

a.

INSERT

ΔMACHINE

$INSERT_{COMBINED}$ [line/newline; spot/cursorspot]

---

status' = SUCCESS $\Rightarrow$
(cursorspot' = cursorspot $\wedge$
newline' = $\langle\rangle$)
status' $\neq$ SUCCESS $\Rightarrow$
ΘMACHINE' = ΘMACHINE

The insertion operation places the new line in the document at the position which appears at the actionline of the screen. The newline component is cleared to allow another new line to be created.

b.

ADD

ΔMACHINE

$ADD_{COMBINED}$ [line/newline]

---

status' = SUCCESS $\Rightarrow$
(cursorspot' = length(doc) $\wedge$
newline' = $\langle\rangle$)
status' $\neq$ SUCCESS $\Rightarrow$
ΘMACHINE' = ΘMACHINE

The add a line operation appends the new line to the end of the document. The newline component is then cleared for reuse. The newly added line will be shown at the actionline of the screen.

c.

DELETE

ΔMACHINE

$DELETE_{EDITOR}$ [spot/cursorspot]

---

status' = SUCCESS $\Rightarrow$
(length(doc) > cursorspot $\Rightarrow$
cursorspot' = cursorspot $\wedge$
length(doc) = cursorspot $\Rightarrow$
cursorspot' = cursorspot - 1 $\wedge$
newline' = newline)
status' $\neq$ SUCCESS $\Rightarrow$
ΘMACHINE' = ΘMACHINE

The delete operation removes the document line visible at the actionline of the screen. If a delete is attempted from a null document, then an ERROR1 status will be returned. A different error condition for this specific problem could have been added at this point but was not for simplicity.

d.

SCROLL

ΔMACHINE

$CURSORMOVE_{MACHINE}$

---

status' = SUCCESS $\Rightarrow$
(doc' = doc $\wedge$
newline' = newline)
status' $\neq$ SUCCESS $\Rightarrow$
ΘMACHINE' = ΘMACHINE

This operation can be used to scroll the screen any amount, either up or down. In order to specify the required user commands we limit the scrolling to one line up or down.

SCROLLUP ≙ SCROLL [amount/-1]

SCROLLDN ≙ SCROLL [amount/1]

e. ADD-CHAR ≙ $\text{ADD-CHAR}_{\text{MACHINE}}$

The new version of ADD-CHAR does not affect any of the state components except as specified in the original version of the operation. Therefore, the operation is trivially promoted to the current state as denoted by the MACHINE subscript.
such as this one.

Although initially specified, the user view changing operations ADDDISPLAY and DELETEDISPLAY have not been promoted since subsequent user requirement clarification has removed the need for them. By leaving these two operations in the abstract design documentation, we have provided a *head start* for likely future system adaptations.

5. The System Module

This level recomposes the designed machine into the remaining aspects of the complete *system* design.

SYSTEM

Basis:

Library: Schema combinators

Prior: MACHINE1

Comments:

In this specification, the abstract machine will be placed into the user's system to complete the design. In this simple case, no changes will be required in the state space but several new observations and operations are required.

Auxiliary Definitions:

a. `possiblekeys is ABSTRACT`

We will not need to explicitly define all of the keystrokes that are possible on our target machine.

b. `legalkeys ≙ chars ∪ {scrollup,`
`scrolldn,`
`insert,`
`add,`
`delete,`
`quit}`

`legalkeys ⊆ possiblekeys`

The keystrokes allowed by our system are explicitly defined and are a subset of all possible keystrokes. Any keystroke not in the set of legal keys will be an error.

State Definition:

Unchanged from MACHINE1.

Observations:

a.

USER

ΦMACHINE

key' : possiblekeys

status' : STATUS

---

key' ∈ legalkeys

status' = SUCCESS

The system user is specified as an abstract relation. Such a view of the user does not prejudice the system against unlikely (but nonetheless possible) user inputs.

BADKEY?

ΦMACHINE

key' : possiblekeys

status' : STATUS

---

key' ∉ legalkeys

status' = ERROR5

where ERROR5 ≙ "The input command is not valid in this system."

If the user inputs an illegal key command, this error condition would result.

USER ≙ LOOP(BADKEY?); USER

The total definition of the user requires that a legal keystroke should ultimately result or the system would never do anything (else) useful.

Operations:

a. SESSION ≙ INITIALIZE; EXECUTE; TERMINATE

The running of a non-continuous (ie. terminating) system is called a *session* of the system. We normally specify that a session is composed of three phases -- initialization, execution, and termination.

b.

INITIALIZE ≙ _____

MACHINE'

---

doc' = ⟨⟩
newline' = ⟨⟩
cursorspot' = 0

The abstract machine initialization must produce a legal machine state for the remainder of the design to be consistent. In more sophisticated systems, part of the machine initialization values may come from the *environment* (eg. disk files, database, etc.) which exists between sessions. For this system, we have designated a starting state with an empty document.

c. EXECUTE ≙ SCREENAPPEARANCE; USER;
LOOP (OPERATE; SCREENAPPEARANCE; USER)

Executing the system requires that: 1) the screen picture be shown to the user, 2) the user select a legal system command, 3) the system transform the state according the appropriate state operations, and 4) the system iterate the previous three requirements until termination. The fourth requirement is the responsibility of the OPERATE operation.

d.

OPERATE

ΔMACHINE

key : legalkeys

---

key ≠ quit

(key ∈ char) ⇒ ADD-CHAR [nextchar/key]
(key = scrollup) ⇒ SCROLLUP
(key = scrolldn) ⇒ SCROLLDN
(key = insert) ⇒ INSERT
(key = add) ⇒ ADD
(key = delete) ⇒ DELETE

This operation represents the driver module of the implemented system. It will execute any legal series of user commands and terminate the session when so directed by the user. Mutual recursion between the OPERATE and EXECUTE operations provides the iteration that was noted in requirement four of the previous paragraph.

e. TERMINATE ≙ MACHINE

The terminate operation will normally define the allowable criteria for termination -- that is, it will specify the machine states in which termination may (or will) occur. The terminate operation will also define which portions of the machine state will continue to exist (in the *environment*) between sessions.

In this simple system, termination may occur in any legal machine state and the system is entirely destroyed at the end of each session.

Although we do not intend to pursue this small example of the methodology to an implementation, it should be obvious at this point the aspects of the design which still must be decided prior to an implementation. For example:

1. how are errors displayed?
2. what is the screen display algorithm?
3. how and where is the cursor to be displayed?
4. how is the new line to be displayed?
5. what is the relationship between physical and logical keys? ...physical and logical screen?
6. which programming language is to be used and (as a consequence of that decision) what will be the implementation structure of the document?

These are the types of decisions (eg. algorithms, user interfacing, non-functional requirements, etc.) which are discussed in the Implementation Plan.

CHAPTER III

A Family of Visible Filing Systems

Prior to applying our proposed software design methodology to the development of an information sharing, windowing screen editor we will present the design for a family of visible filing systems. This abstract design is typical of an entry in our proposed library of high level design abstractions. This high level design will help us intelligently decompose and plan the design strategy for the case study which follows in Chapters IV and V.

III.1 Introduction.

Computers are commonly used to store documentary information, any part of which can be made visible on a CRT screen or printing device. The software system which provides this information display is called a visible filing system. Some systems also allow the viewing screen to be partitioned so that various documents can be made visible in any desired arrangement on the screen. Such a system is called a windowed screen display manager. Other systems allow information (figures, text, etc.) to be shared between documents in a convenient and manageable manner. These systems are called information sharing filing systems. The systems we are going to specify here are information sharing, windowed, visible filing systems.

This family of display and filing systems is an important one to define in a reusable and consistent manner because it has a very wide applicability but is not yet totally understood or exploited. As [Meyrowitz,81] puts it:

"The window manager is an emerging computing paradigm which allows the user to create multiple terminals on the same viewing surface and to display and act upon these simultaneous processes without loss of context.... Most [current] implementations and their associated designs are not readily available for common use; extensibility is minimal."

Several attempts have been made at defining such a system: [Carlson,82; Guttag,82; Hoare,81; Horning,79; Mallgren,82; Meyer,82;

Meyrowitz,81; Richardson,81; Sorensen]. Many of these are too concretely specified to be reusable or extendable to other design efforts. Few allow information sharing between documents -- an important aspect of our system. The development presented here is based upon [Hoare,81; Richardson,81] and represents a very flexible family of windowed filers with information sharing and (potentially) graphical capabilities.

III.2 Synopsis.

Sharing information between documents one has created and stored is a very useful but potentially very complex concept. The main advantage of sharing information is that a document change can be propagated to every other occurrence of the shared information. On certain document areas, which we will call the *masters*, the information has been printed and can be seen immediately. Other documents areas, called the *slaves*, contain only *references* to the information on the master document areas. These references must specify where the information is to appear in the slave area as well as what information is to appear there. Of course, the information appears in the proper position of a slave document area only if we *look up* the referenced information on the master area. This type of referencing is commonly done in books: for example, with footnotes specifying where the master material is to be found and a special symbol (eg. a footnote number) marking the spot where the shared information should appear (after look up) in the text.

⋮

THE FULL RANGE OF SOFTWARE ENGINEERING PROBLEMS.[+] EACH FIGURE MUST

⋮

[+] SEE [SWARTZ, 63], PAGES 20-23, FOR THE COMPLETE LIST OF PROBLEMS.

FIG. III-1 -- A Reference.

Before discussing further the concept of information sharing, it is necessary to define how we represent the information in our system. Consider a filing cabinet containing many large sheets of colored paper. Each sheet of paper is a document and is given a unique name. On each of these documents is a coordinate grid so that the location of every *speck* of information on each document can be identified by a coordinate. In other words, a reference to the filing cabinet with any coordinate location on a specific document should show us what speck of information *appears* there.

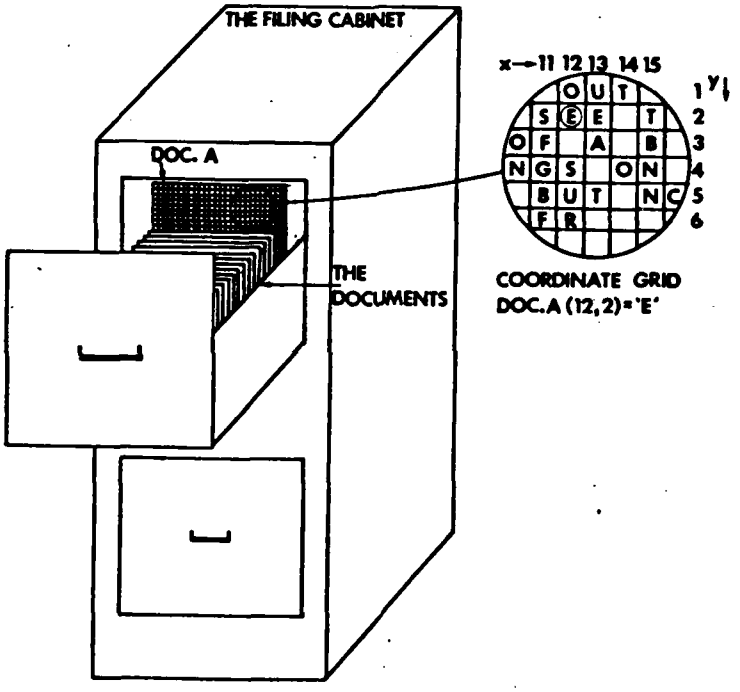


FIG. III-2 -- The Filing Cabinet.

Of course, we should be able to find the appearance at each document coordinate of our filing cabinet and it should not matter whether a reference to shared information defines the appearance or if the appearance is immediately available. Therefore, we need a technique for looking up a reference one if the coordinate we wish the appearance of is a slave. The technique used is called a *projection.* A projection is a function which specifies a master coordinate for each point in a slave. The appearance of any coordinate which is sharing information with a master coordinate must be the appearance of that master coordinate; that explains how changes to shared information are propogated to all slaves.

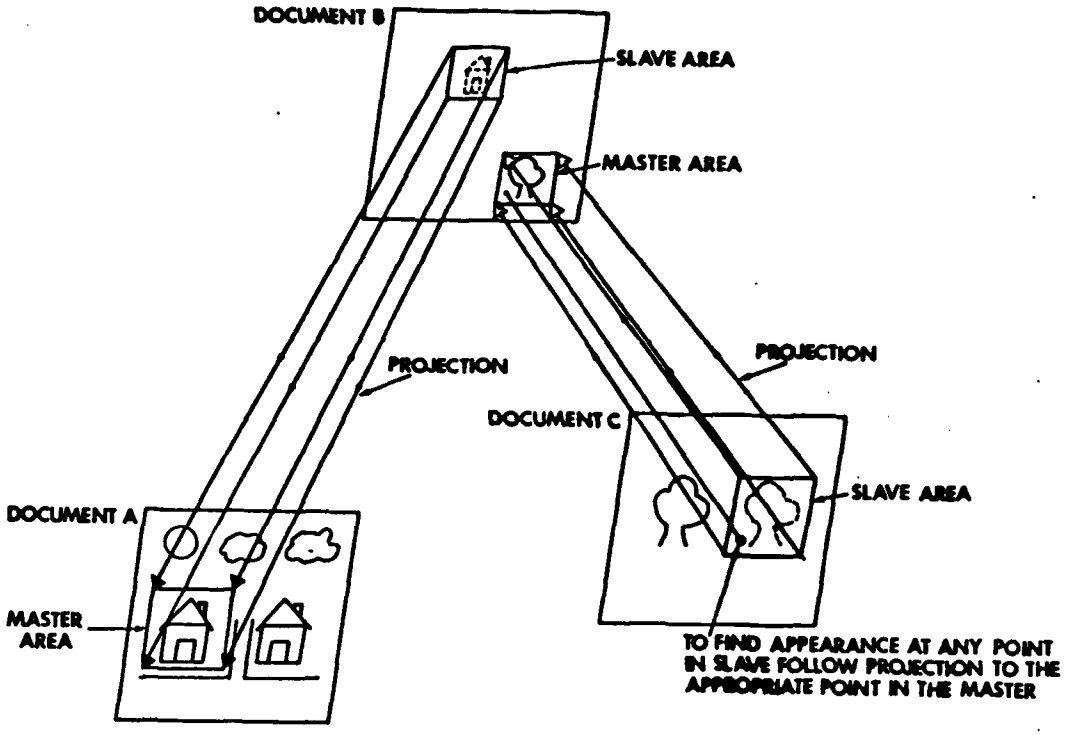


FIG. III-3 -- Projections.

So far, the system is simply a filing cabinet which contains all of the information we have created and stored, and a projection to act as look up instructions if there is shared information in one of our documents. But what if we need several different look up instructions for different information sharing activities? Certainly, there is nothing to prevent us from having numerous projections, P_1, P_2, P_3, . . . , P_n, for all of the different slave/master sharing arrangements we require in the filing cabinet. The only complication arises when two or more of these projections provide look up instructions for a particular coordinate -- which instructions should be heeded in that case? The solution is to list the projections in order of precedence so that it will be obvious which projection instruction must be followed if there is a choice. For example, with the list of projections $\langle P_1, P_2, P_3, P_4, P_5, \ldots \rangle$, and if a coordinate location appears in the slave of both P_2

and P_5, then P_5 is the look up instruction which will be used for that location since it appears later in the list.

How, then, do we maintain the precedence relationship we have developed in the projection list and still produce the single projection needed as a look up instruction to find the information appearing at any point in the filing cabinet. The answer is to *flatten* the list of projections so that only the look up instructions prescribed by the projection with the greatest precedence is included for each coordinate. It should be noted that reordering of the projection list may significantly change this flattened projection and, therefore, the appearance of the entire filing cabinet.

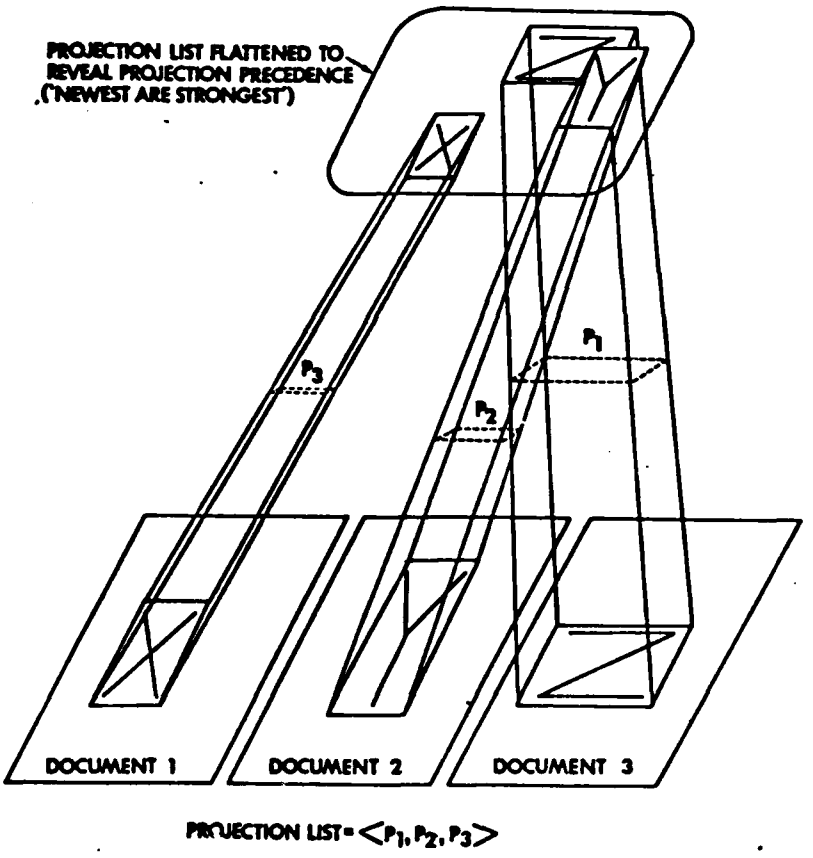


FIG. III-4 -- Flattened Projection List.

But there still may be a problem -- if the master of a projection is also the slave of a projection in the list, then a single application of the flattened projection will not give us the appearance we desire. That is, if in looking up one reference we find a further reference, then we must also have look up instructions for that second reference.

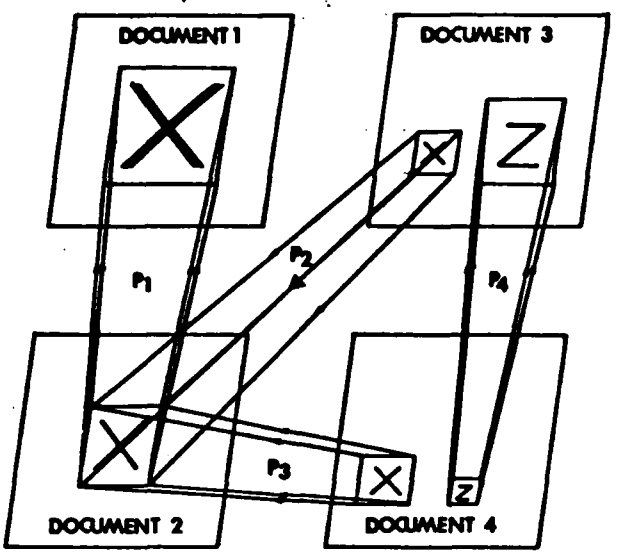


FIG. III-5 -- A Projection Network.

This time the solution to our problem is to apply the flattened look up instructions the exact number of times required for the coordinate whose appearance is desired. So, with our filing cabinet full of information and with that information being shared in any desired arrangement, we are able to find the appearance of any point on any document in the filing cabinet. The only useful capability which we do not now have is changing the appearance of information as it is being shared. For example, it might be useful to italicize information that is shared from another document to show that it is not original information. This can be done quite easily with a *filter*. A filter will replace any possible speck appearance with a desired new appearance according to the user's orders. Each slave in the flattened projection can be viewed through its own filter so that any shared information will appeared altered as appropriate.

Two final points should be noted here. First, there is the possibility that the projection network will result in a infinite series of look up instructions for certain coordinates (eg. when two documents share information mutually between themselves). When this happens, there is no true master and the appearance of these coordinates is undefined just as you would expect them to be.

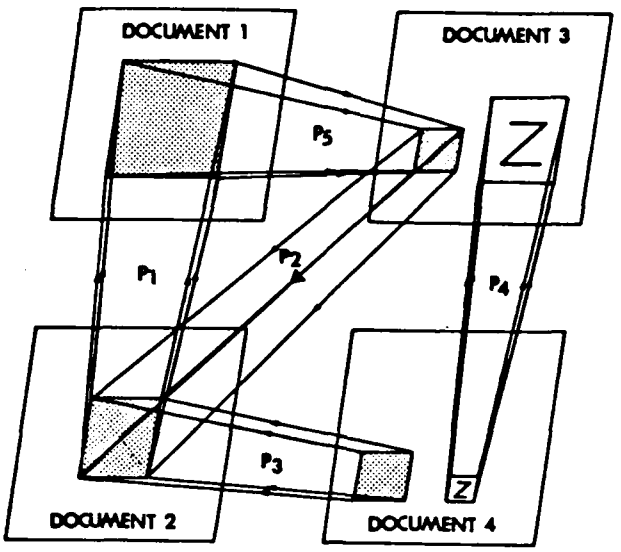


FIG. III-6 -- Undefined Appearance.

The second point to be made is that it is possible to change the size or shape of the shared information by having the projection cause scaling or skewing of the information.

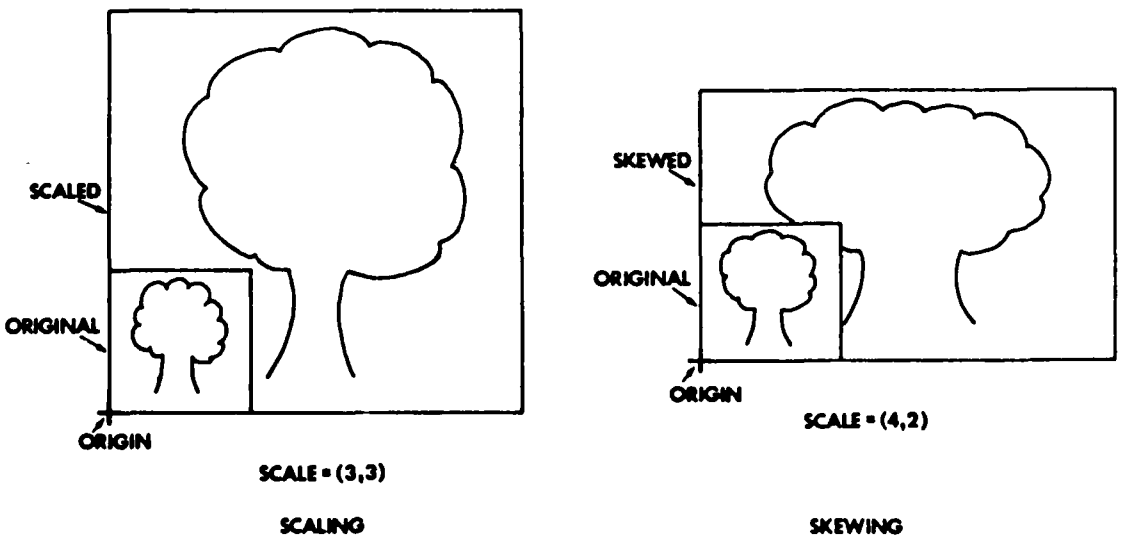


FIG. III-7 -- Scaling or Skewing.

We can now summarize the information sharing aspects of the filing system. A piece of information which is shared among documents actually only exists in one place, called the master. All other documents which share

this information (slaves) have only a reference back to the master document. This reference is in the form of a projection which defines the relationship between the master area and the slave area. The relationship may be a straightforward sharing of the unadulterated information or may include some complex scaling to change the size, shape, and appearance of the information for the slave. Moreover, the relationship of the master and the slave (by way of the projection) may even involve the selective but consistent modification of the content of the shared information by placing filters on the projection. Obviously, any change to a master area results in an equivalent change to all of its slave areas.

Basic to a windowed display system is the capability to position parts of various documents in any arrangement on the screen. However, this is just another form of the problem of infomation sharing between documents and is solved with the same mechanism -- a projection. The slave area of this projection will always be the screen, while the master area will be in the filing cabinet. Of course, the viewing system can display only that which the filing system has stored.

Because this model is the basis for a family of filing/viewing systems, the emphasis will be on generality above other considerations; consequently, almost all design decisions will be postponed to later levels of specification. For example, one member of this family of systems might produce documents entirely from *projections* of basic, character documents (ie. letters, numerals, etc.) onto blank pieces of paper; while other members might allow characters to actually be *typed* onto the pieces of paper. As a further example, one member of the family may require what is seen on the screen to come from a single "viewing" document in the filing system, while some others may allow the screen to have the selection of all documents with which to share information.

III.3 Specification.

FILER1

Basis:

Forward: FILING CABINET1.1 / PROJSET1.2 / DISPLAY1.3

Comments:

This specification represents a windowed, information sharing visible filing system.

(The FILER specification is adjourned so that the FILER components: FILING CABINET, PROJSET, and DISPLAY, can be specified. This brief introduction to FILER indicates the hierarchical nature of this module specification.)

FILING CABINET1.1

Comments:

The FILING CABINET models the set of user documents which contain all the information stored and displayed in the system. Each document is separate and distinguishable from all other documents and is allowed its own color of paper.

Auxiliary Definitions:

a. speck is ABSTRACT

b. coordinate is ABSTRACT

In the interest of generality, the two basic concepts speck and coordinate are left only informally defined: a coordinate is a point in space at which a speck is visible. For example, on the surface of a flat screen, a coordinate could be defined by a pair of real numbers x and y, where x measures the distance from the top of the screen and y measures the distance from the left edge. On a simple screen, the speck may be just one of the two values black or white. On a color screen, it may be a triple of real numbers (r,g,v) indicating the intensity of the colors red, green, and violet. On a document containing only text, a coordinate might be a pair of integers (l,c) giving a line number and column number, and a speck might be one of a limited range of characters that could appear in any line and column of the document.

c. surface ≜ coordinate ⇸ speck

A surface is defined by giving the speck which appears at each coordinate of the surface. The domain of the function gives the extent of

the surface. For example, the surface of a CRT screen may extend over all x and y coordinates in the range -1 to 1, and at each of these points, the illumination is defined by the corresponding speck. A pictorial document is likely to have a very large domain since specks at many different points will be required to define even a simple figure. On the other hand, a textual document, which could be defined as a mapping from coordinates to the letters which occupy each row and column, will likely have a smaller, simpler domain. Any surface defined in the state space could have an infinite domain.

d. docname is ABSTRACT

Again in the interest of generality, the set of names used for the documents in the filing cabinet is left undefined. The only requirement is that document names be unique.

State Component Definition:

FILING CABINET1.1

documents	:	docname ⇸ surface
colors	:	docname ⇸ speck
filecolor	:	speck

dom (documents) = dom (colors)

This formal definition says that the FILING CABINET consists of of surfaces (informally called documents) each of which can be accessed by its unique name. These documents each have a specific color of paper on which the information is presented. Finally, the part of the filing cabinet not filled with documents has a color of its own.

The *selection* of documents which exist in the filing cabinet is:

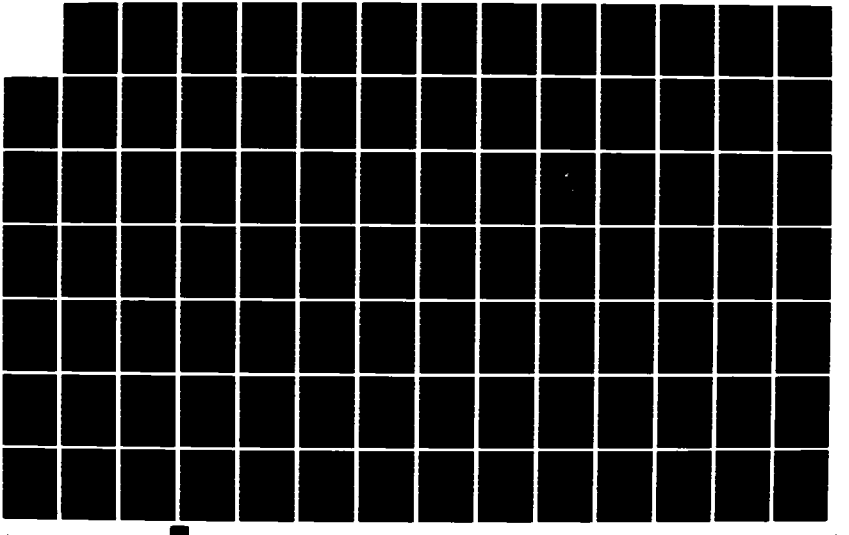
dom (documents).

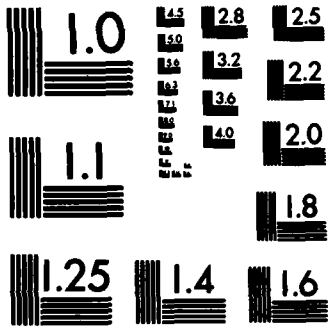
The *extent* (size and shape) of a document listed in the selection of documents which has the name dn is:

dom (documents (dn)),

AD-A132 569 FORMAL TECHNIQUES IN THE MANAGEMENT OF SOFTWARE DESIGN 2/4
(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH
W E RICHARDSON 17 JUN 83 AFIT/CI/NR-83-28D

UNCLASSIFIED F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

while the *content* of that document is:

ran (documents (dn)).

The *appearance* at a coordinate c on this document dn is:

documents (dn) (c).

(Later we will define the appearance of any coordinate outside the extent of document dn but on the document's *colored paper* to be determined by colors(dn). For a document name which is not in the index of documents and hence not in the filing cabinet (dn $\notin$ dom (documents)), the appearance of any coordinate will be the filing cabinet color, filecolor.

Operations:

The following operations on the FILING CABINET allow the addition or deletion of a document or the changing of a background color. These operations have been kept abstract by not specifying, for example, how the new document is actually created.

a.

ADD-DOCUMENT

ΔFILING CABINET1.1

dn : docname

sf : surface

sp : speck

---

dn $\notin$ dom (documents)

documents' = documents ∪ {dn $\mapsto$ sf}

colors' = colors ∪ {dn $\mapsto$ sp}

filecolor' = filecolor

Given a new document, its unique name, and the color of paper on which it is to appear, the filing cabinet is updated with this new information by ADD-DOCUMENT.

b.

DELETE-DOCUMENT

ΔFILING CABINET1.1

dn : docname

---

documents' = documents \ {dn}

colors' = colors \ {dn}

filecolor' = filecolor

Any document and its associated paper color can be deleted from the filing cabinet by supplying the name of the document to be deleted.

c.

COLOR-CHANGE

ΔFILING CABINET1.1

dn : docname

f : speck → speck

---

dn ∈ dom (colors)

documents' = documents

colors' = colors ⊕ {dn ↦ f (colors (dn))}

filecolor' = filecolor

Any existent paper color can be changed by the application of a changing function (filter) to it. Of course, the name of the document whose paper color is to be changed must be supplied to this operation along with the filter.

d.

FILECOLOR-CHANGE

ΔFILING CABINET1.1

f : speck → speck

---

documents' = documents

colors' = colors

filecolor' = f (filecolor)

The file background color can also be changed with a filter function.

PROJSET1.2

Basis:

Prior: FILING CABINET1.1 (Auxiliary definitions)

Comments:

The PROJSET models the interactions or sharing of information between documents in the FILING CABINET. The sharing is done by projecting one area (the master) onto another (the slave). The order of the projections is significant because it is used to determine the result where several masters have projected into the same slave area (projection overlap). Projections can scale or skew information so that the size, shape, or appearance of a document part can be changed as it is being shared. The information being projected can also be filtered. The filtration is done on the slave end of the projection and changes the content of the area being projected.

Projections will also be used to share information between the filing cabinet and the *screen*, thereby providing the capability to define a windowed display system.

Notice that projections are defined in what at first glance may seem to be a backward fashion (from slave to master). The reason for this form of projection specification becomes clear when you realize that the important requirement will be to determine the speck which *appears* at a given coordinate. It will seldom be necessary to ask at which coordinate a speck will appear.

Auxiliary Definitions:

a. `filter ≙ speck → speck`

The filter is used to change the specks on a surface (eg. exchanging

red and green, or changing characters to reverse video.) Notice that the filter is a total function on speck.

b.

COORD

dn	:	docname
coord	:	coordinate

This definition allows the document name to be used as a component of the coordinate in an expanded coordinate system. That is, any COORD specifies a coordinate on a particular document in the filing cabinet.

c. projection ≙ COORD ⇸ COORD

A projection is used to map or *project* each coordinate of a slave surface area onto some master surface area. Consequently, it gives the look up instructions for references where information is being shared between documents. For example, if P is a projection, then P(c) is the master coordinate whose appearance is given to slave coordinate c.

With this powerful device, a document can be made to share information with any document in the filing cabinet. It is also the projection which allows us to display the filing cabinet information on the screen in any arrangement that we desire.

$$slave : projection \rightarrow \mathbb{P}\ (COORD)$$
$$slave \triangleq dom$$

$$master : projection \rightarrow \mathbb{P}\ (COORD)$$
$$master \triangleq ran$$

The projection also defines the scaling or distortion between the master and the slave. No restriction is given on the *size* and *shape* of either the master or the slave. Notice that several different documents can be mapped using a single projection.

d. filter-apply ≜ COORD → filter

A filter-apply will exist for the slave of each projection. It determines which filter is to be applied to change the appearance of information being shared between documents. Every coordinate in the slave of a projection will appear to contain the speck derived from application of both the projection and the appropriate filter. Of course, one possible filter is the identity function, I(speck), which has no effect on the projected appearance.

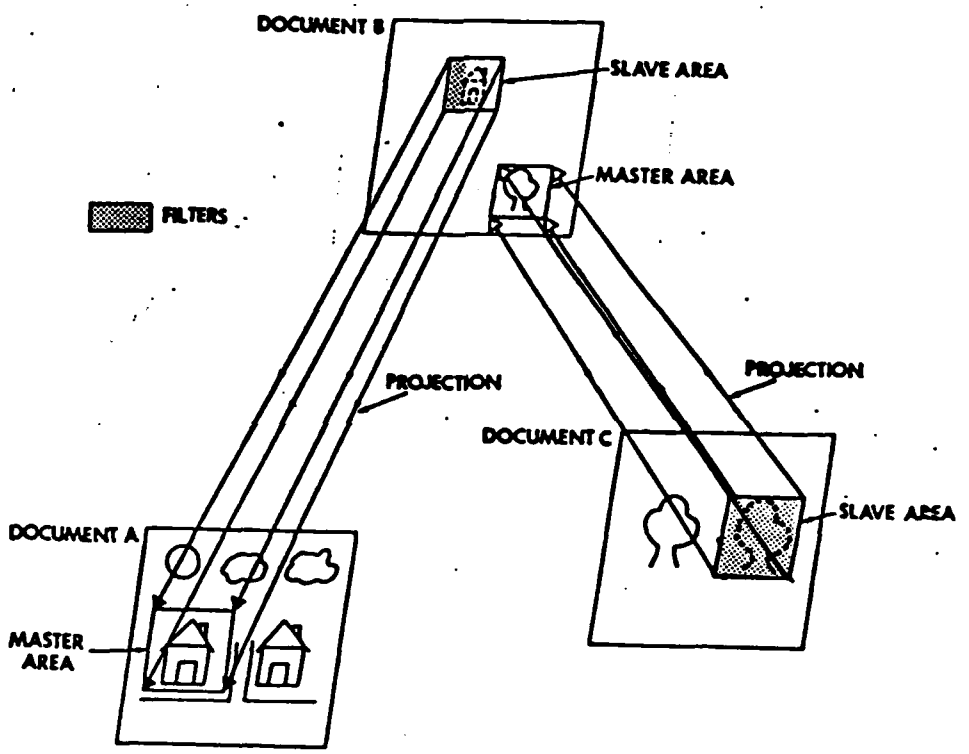


FIG. III-8 -- Projections and Filters.

State Component Definition:

PROJSET1.2

```
projlist   : SEQ[projection]
filterlist : SEQ[filter-apply]
```

```
dom (projlist) = dom (filterlist)

(∀x : dom (projlist))
      (dom (projlist(x)) = dom (filterlist(x)))
```

PROJSET contains a list of projections and a list of filter-applys. For any projection number n in projlist, filterlist(n) is the appropriate filter-apply function.

Notice that the second predicate suggests that the appropriate filter is to be applied to the slave of the projection. Again, the reason is that it will be important to know what speck (or filtered speck) appears at each coordinate point.

A question to be answered now is how overlapping projection slaves will be resolved. The simplest solution requires a function which will *flatten* the list of projections into a single projection. Of course, since projection is a function, there is no ambiguity with a single projection. The standard function we will use to do the required *flattening* is called update:

$$update\ (v) = v\ (1)\ \textcircled{0}\ v\ (2)\ \textcircled{0} \ldots \textcircled{0}\ v\ (length(v))$$

where the overriding operator "0" means:

$$r1\ \textcircled{0}\ r2 = r1 \setminus dom(r2) \cup r2 \qquad (r1 \text{ and } r2 \text{ are relations})$$

Therefore, using update the later projections in the list will override earlier projections in the case of overlap. The order of the projections in the projlist can have a significant effect on the appearance of the documents in the filing cabinet or as seen on the screen. From the definition of add-projection below it is obvious that newly added projections take precedence over the other projections.

The updated projection list gives us a single projection so slave overlap is no longer a problem, but we still face the question of how to traverse the projection network so that we can find the *ultimate master* for any input coordinate. This is also easily resolved with one of our standard tools, in this case loop. Loop has the following properties:

$$loop(R) = (R^*) \,/\, dom(R)$$

$$(\mathbb{I}(dom(R)) \cup R);\ loop(R) = loop(R)$$

Therefore, loop will apply the flattened projection as many times as required to find the ultimate master of a projected coordinate. A caution must be given here: loop(update(projlist)) may not be defined for all

coordinates in the slave of the various projections in projlist. The reason for the undefined ultimate master in these undefined cases is circular projection paths in the flattened projlist. This is the same problem posed by placing two or more mirrors in such a manner that they mutually reflect--certain points seen on one mirror may never be traceable through all the reflections to their source (ultimate master).

Operations:

The following operations on the PROJSET cause the addition, deletion, or swapping of documents or the changing of a filter. Like the operations on the FILING CABINET, these operations have been kept very general.

a.

ADD-PROJECTION

ΔPROJSET1.2

p : projection

fa : filter-apply

---

dom (p) = dom (fa)

projlist' = projlist * ⟨p⟩

filterlist' = filterlist * ⟨fa⟩

A projection (and its associated filter-apply) is added to the end of the appropriate list if the domain of the new filter-apply is the slave of the new projection.

b.

DELETE-PROJECTION

ΔPROJSET1.2

m : $\mathbb{N}_1$

---

m ≤ length (projlist)

projlist' = delete (projlist,m)

filterlist' = delete (filterlist,m)

Any existing projection and its filter-apply can be deleted by specifying the number of the projection to be removed.

c.

SWAP-PROJECTION

ΔPROJSET1.2

n : $\mathbb{N}_1$

m : $\mathbb{N}_1$

---

n $\leq$ length (projlist)

m $\leq$ length (projlist)

projlist' =
projlist $\oplus$ {n $\mapsto$ projlist (m)} $\oplus$ {m $\mapsto$ projlist (n)}

filterlist' =
filterlist $\oplus$ {n $\mapsto$ filterlist (m)} $\oplus$
{m $\mapsto$ filterlist (n)}

Swapping two projections reverses their order in the list and likewise moves the two associated filter-applys. Since the order of projections in the projlist is very important in determining the appearance of a document, the ability to swap two projections is required.

d.

SWITCH-FILTER

ΔPROJSET1.2

n : $\mathbb{N}_1$

f : filter

C : $\mathbb{P}$(COORD))

---

n $\leq$ length (filterlist)

projlist' = projlist

filterlist' = filterlist $\oplus$
{n $\mapsto$ filterlist (n) $\oplus$ (f $\circ$ filterlist (n)) $\triangleright$ C}

This operation allows the filter for part of the projection slave to be changed. Note that the change is by composition rather than replacement.

DISPLAY1.3

Basis:

Prior: FILING CABINET1.1 (Auxiliary definitions)

Comments:

DISPLAY specifies what coordinates of which documents will be displayed on the screen.

State Component Definition:

DISPLAY1.3

seearea	:	$\mathcal{P}$(COORD)

DISPLAY is the set of coordinates whose appearance the user wishes to see *displayed* on the terminal screen.

FILER1

(The FILER specification, which was adjourned to allow the substate components to be defined, can now be restarted.)

Auxiliary Definitions:

a.

$$\text{background} : X \rightarrow \text{COORD} \rightarrow X$$

$$\text{background} \triangleq (\lambda x)\ (\lambda c)\ x$$

This constant function *spreads* the background value (eg. *color*) to all coordinates.

State Definition:

FILER1

FILING CABINET1.1
PROJSET1.2
DISPLAY1.3

The FILER, then, models the complete filing/viewing system by combining the three components. FILING CABINET and PROJSET specify a very abstract information sharing filing system. All three components will be used to specifiy a simple abstract windowed display system.

Observations:

Now it is possible to determine the appearance of the information in the filing cabinet and on the user's screen. This is done by creating the following functions to integrate the FILER components.

a.

FILE-APPEARANCE

ΦFILING CABINET1.1

view' : COORD ⇸ speck

(∀C : COORD | dn (C) ∈ dom (documents))
(view' (C) = (background (colors (dn (C))) ⨾
uncurry (documents)) (C))

This function *places* the documents on their background paper colors. Given a FILING CABINET, any coordinate on any existing document will determine a speck. Any coordinate which is not in the extent of the document (ie. not in the domain of the document) will appear as the appropriate background speck.

Remember,

documents ≙ docname ⇸ (coordinate ⇸ speck)

and

uncurry (documents) = (docname × coordinate) ⇸ speck
= COORD ⇸ speck

b.

PROJECTED-APPEARANCE

ΦFILER1

view : COORD ⇸ speck

prjview' : COORD ⇸ speck

---

prjview' = (background (filecolor) ⨾ view)
∘ loop (update (projlist))

This function adds the shared information look up instructions defined by the projections to the documents. The projection list *flattened* by update will be applied repeatedly by loop until a document coordinate

outside the domain of the flattened projection list is produced. (If the loop *does not terminate, the result is undefined.)* *This coordinate will use the* document appearance of the previous function to determine its speck if the coordinate is on an existing document or its paper. If the coordinate is not on one of the papers in the filing cabinet, then it will contain the filing cabinet *background color.*

c.

```
APPEARANCE_____________________________________________
      ΦFILER1
      prjview      :      COORD ⇸ speck
      fileview'    :      COORD ⇸ speck
__________
      fileview' (C) = ((background (I(speck)) ⊕
                           update (filterlist)) (C)) ∘
                              prjview (C)
_______________________________________________________
```

Finally, this function determines the appearance of the entire filing cabinet. It applies the appropriate `filter` to each `speck` produced by the `PROJECTED-APPEARANCE`. If the given coordinate is not an element of a `projection` domain, then the coordinate will likewise not be a element of a `filter-apply`. In this case, the identity function, *I*(speck) will leave the speck unchanged. (Remember that the identity function, *I*(COORD) is *built into* `loop`.)

d.

```
DISPLAY-APPEARANCE_____________________________________
      ΦFILER1
      fileview     :      COORD ⇸ speck
      picture'     :      COORD ⇸ speck
__________
      picture' = fileview ↾ seearea
_______________________________________________________
```

This last observation allows us to restrict our view of the filing cabinet *to those areas which are of interest to the user. We now have the ability*

to formally specify the picture that the user will see displayed on his screen:

DISPLAYED ≙ FILE-APPEARANCE; PROJECTED-APPEARANCE;
APPEARANCE; DISPLAY-APPEARANCE

We should again note that this is a partial function since no attempt has been made to restrict circular projection networks from the PROJSET.

Operations:

The operations defined below will not be presented as FILER to FILER functions. The reason for this is quite simply to delay unnecessary promotion (and, hence, complication of simple functions) and to put off premature design decisions. For the same reason, the operations previously defined on the various components of the state will not all be promoted to state to state functions.

The projection concept is a very powerful one. We shall now use projections as a convenient means of changing parts of the state in addition to their previous use as a state component in the PROJSET. The projections used to change parts of the abstract machine state we will refer to as *updating projections* to indicate that they do not (necessarily) coincide with the state component projections. We can now make state changes by composing part of the state with an updating projection. The advantage of composition changes rather than outright replacement of state components is that it makes reversing (ie. undoing the effects of) the operation possible. The following table summarizes the operations which can be developed from state components and updating projections:

1. move:
   - a projection (frame, image, or both)
   - an area of filter application
   - a document (part)
   - an area seen on the screen
2. change the size/shape of:
   - a projection
   - an area of filter application
   - a document (part)
   - an area seen on the screen

a.

change-slave,

change-master : projection × projection → projection

change-slave (up,p) ≙ p ∘ up

change-master (up,p) ≙ up ∘ p

change-projection : projection × projection ⇸ projection

change-projection ≙ (λup,p | up ∈ partial-injection)

$$up \circ p \circ up^{-1}$$

This will allow the slave and/or master of a projection in PROJSET to be moved, changed in size or shape, scaled or skewed.

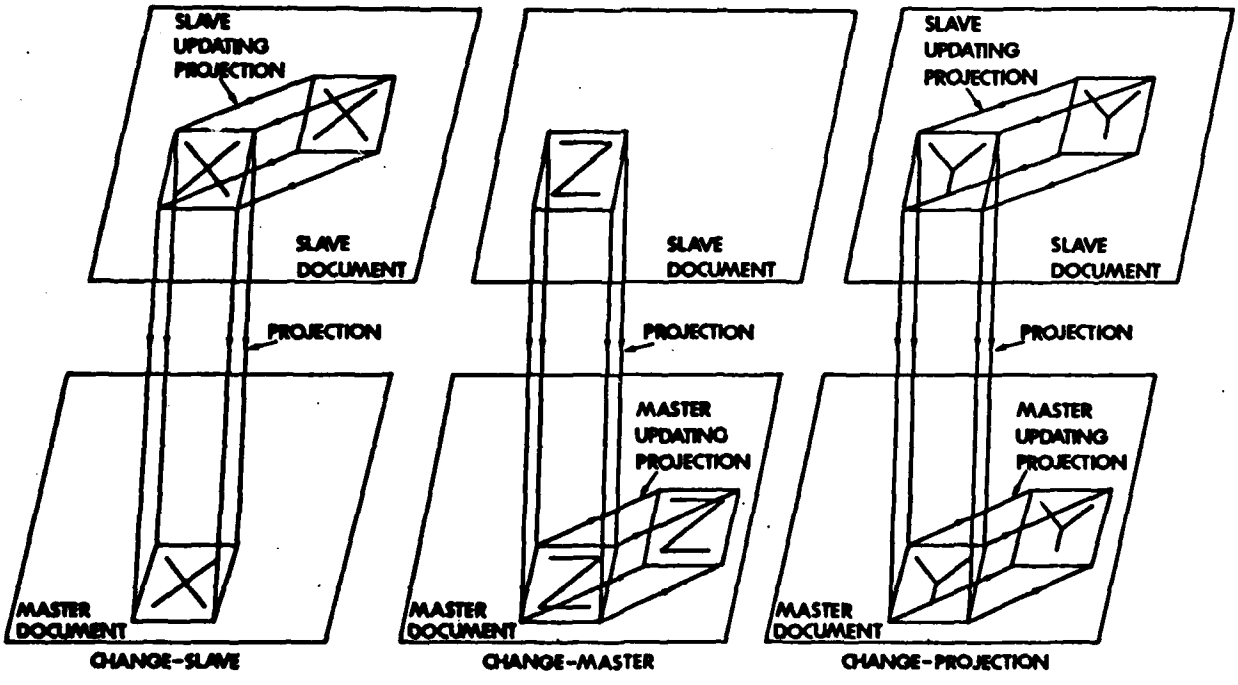


FIG. III-9 -- Projection Changing.

b.

change-filter-area : filter-apply × projection →

filter-apply

change-filter-area ≙ op(∘)

We can change the area of application of the filter(s) associated with a projection.

c.

$$\text{change-document} : (\text{docname} \nrightarrow \text{surface}) \rightarrow \text{projection} \nrightarrow (\text{docname} \nrightarrow \text{surface})$$

$$\text{change-document} \triangleq (\lambda S)\ (\lambda C : \text{COORD} \mid \text{dn}(C) \in \text{dom}(S))\ (\lambda D : \text{COORD} \mid \text{dn}(D) \in \text{dom}(S))$$

$$S \oplus \{\text{dn}(C) \mapsto (S(\text{dn}(C)) \oplus (S(\text{dn}(D)) \circ \{\text{coord}(C) \mapsto \text{coord}(D)\}))\}$$

This function allows the projection (movement) of a document (part) and can thereby change the size or shape of the slave document. It does not affect the master document area or allow the creation of a new document.

d.

$$\text{change-display} : \text{projection} \times \mathbb{P}(\text{COORD}) \rightarrow \mathbb{P}(\text{COORD})$$

$$\text{change-display} \triangleq (\lambda P, s)\ P\ (s)$$

This changes the document areas displayed on the screen.

FILER2

Basis:

Prior: FILER1

Comments:

This specification restricts the abstract coordinate system of FILER1 to a two dimensional Cartesian system. This restriction recognizes the obvious choice of coordinate systems for our (textual) documents since a document is normally a flat, two dimensional object and since the mathematical basis of a Cartesian space is widely understood.

Requirements / Design Decisions:

[[1]] The coordinate system is two dimensional.

Auxiliary Definitions:

Add the following to the current theory:

PAIR

Z

$x : Z$

$y : Z$

+

Z

A,

B,

$C' : PAIR$

$Z \subseteq INT \vee Z \subseteq real$

$x(C') = x(A) + x(B)$

$y(C') = y(A) + y(B)$

Furthermore, "-" and "*" are similar to "+".

Z

A,
B,
C' : PAIR

$Z \subseteq \mathbb{INT} \vee Z \subseteq real$

$C' = x(A) < x(B) \vee (x(A) = x(B) \wedge y(A) < y(B))$

The other lexicographic relational operators are similar.

The overloading of operators should not cause confusion since the context will easily determine which operator is intented.

b. coordinate ≙ PAIR [real] [[1]]

Here we have defined our previously abstract notion of coordinate.

State Definition:

```
FILER2 [speck, docname] ≙
        FILER1 [coordinate ≙ PAIR [real], speck, docname]
```

We have restricted the state space to use a coordinate plane which is two dimensional.

FILER3

Basis:

Prior: FILER2

Forward: PROJSET2.2 / DISPLAY2.3

Comments:

This is the first step in limiting the very abstract family of systems. This level will introduce various design decisions which are large in scope and effect. The most significant of these decisions is the definition of the coordinate system for the documents and the screen as two dimensional, rectangular, and integer. With this decision, it was also concluded that scaling and skewing of information is not required in this system.

Other decisions limit the power of projections by 1) requiring that the slave and master of any projection each be confined to a single document paper, and 2) limiting the allowable filters to one per projection in addition to the identity filter. Finally, the display is to show only a single document selected from the filing cabinet.

These decisions were taken because they typify a certain class of systems: those which deal mainly with textual documents, not requiring high flexibility of document design in shape and appearance. Of course, a large number of systems still fit into the family defined by this level of specification.

PROJSET3.2

Basis:

Prior: PROJSET1.2

Comments:

This specification simplifies the projections by requiring that slave and master of a projection must define areas on single documents. This in no way diminishes the power of the set of projections but does restrict the power of each individual projection. The second restriction presented here will allow only a single area of each projection to be modified by a filter. Again, this does not reduce the power of the whole projection set but does simplify individual projections.

Requirements / Design Decisions:

[[1]] The slave and master of a projection must each be an area on a single document.

[[2]] Each `filter-apply` can use only one `filter` besides the identity.

State Component Definition:

PROJSET3.2

PROJSET1.2

[[1]] $(\forall x : \text{dom}(\text{projlist}))\ (\exists a,b : \text{docname})$
$(\text{dom}(\text{projlist}(x)) \subseteq (\text{COORD} \mid \text{dn} = a) \wedge$
$\text{ran}(\text{projlist}(x)) \subseteq (\text{COORD} \mid \text{dn} = b))$

[[2]] $(\forall x : \text{dom}(\text{projlist}))\ (\exists f : \text{filter})$
$(\text{ran}(\text{filterlist}(x)) = \{f\} \cup \mathbb{I}(\text{speck}))$

The appropriate restrictions are added to the component state space definition.

DISPLAY3.3

Basis:

Prior: DISPLAY1.3

Comments:

Our current design gives us the option of seeing parts of a variety of different documents or of selecting a single document to view at any particular time. Of course, in the latter case the selected document could be a complex mosaic of parts of the other documents.

We will choose this latter display technique because it simplifies the definition of the relationship between what the user wishes to see (seearea) and the form of that area presented on the screen. That is, with this definition, the screen is simply an area from one of the documents in the filing cabinet.

Requirements / Design Decisions:

[[1]] The screen displays part of a single document at one time.

State Component Definition:

DISPLAY3.3

DISPLAY1.3

[[1]] $(\exists x : \text{docname})$
dn (seearea) = {x}

FILER3

Requirements / Design Decisions:

[[1]] The coordinate frame is two dimensional, rectangular, and integer.

State Definition:

```
FILER3_____________________________________________________
     FILER2 [coordinate = PAIR [ INT ]]
            (PROJSET1.2 ⊳⊲ PROJSET3.2;
             DISPLAY1.3 ⊳⊲ DISPLAY3.3)
```

The definition of the type coordinate is restricted to pairs of integers from its previous definition of pairs of real values. The remainder of the state space is changed as previously specified in the state component definitions.

CHAPTER IV

The Abstract Baseline

This case study applies the techniques of the proposed software development methodology in a limited and controlled demonstration. The case study project is a moderately sized, single person development which assumes the existence of a typically naive client. Since this study represents the first test of a potentially complex methodology, it is appropriate that the *degrees of freedom* of the project be limited by virtually eliminating the interactions (with clients, management, design team members, implementors, etc.) that would exist in a practical development. In actuality, the portions of the methodology skipped by this initial controlled project (eg. configuration management, resource management and estimation, verification and validation, etc.) are generally aspects which are *imported* from other design strategies and, hence, are not unique to this methodology. The case study does emphasize the aspects which uniquely form the kernel of the methodology: eg. the structuring of the formal specification, the use of baselines within the formal specification, documentation of the limitative design technique, the use of very abstract starting points, etc.

It is important to remember, however, that the methodology was designed for large, multi-person software projects. Therefore, results of this scaled down case study must ultimately be extrapolated to the large development where communication among participants and management control are of the highest priority for an efficient project. It is only at this large scale and over a full life cycle (the case study does not attempt to simulate product testing, handover, operation, or maintenance) that the true cost reduction benefit of this methodology can be determined.

This chapter contains a portion of the abstract baseline documentation. This documentation includes the informal requirements

definition, the problem decomposition, the formal abstract design specification (references), and the provisional user's manual.

IV.1 Requirements Definition.

The object of this case study is an information sharing, windowed screen editor for a personal computer system. The requirements of the desired system are given informally and only very generally by the following guidelines:

(1) Only those parts of the textual information that are currently visible can be updated. All the effects of each update are immediately visible, and can be immediately undone.

(2) All textual information in the system can be brought into view and updated by uniform methods. There are no specialized techniques necessary for manipulating file names, directories, dates, version numbers, etc. All these are documentary information, accessible and updatable in the same way.

(3) The editing commands should be simple, low risk commands which do not rely on a hierarchy of modes.

(4) All user interfaces to the system should be consistent and designed to reinforce the user's confidence in and understanding of the system.

It is assumed that no firm information is available about the hardware to be selected for the customer's system until such a selection is required for further progress. However, from the nature of the case study system it is (reasonably) presupposed that the hardware would include a screen with limited or no graphical capabilities, and whose character set could be stored encoded in the memory of the computer.

IV.2 Decomposition.

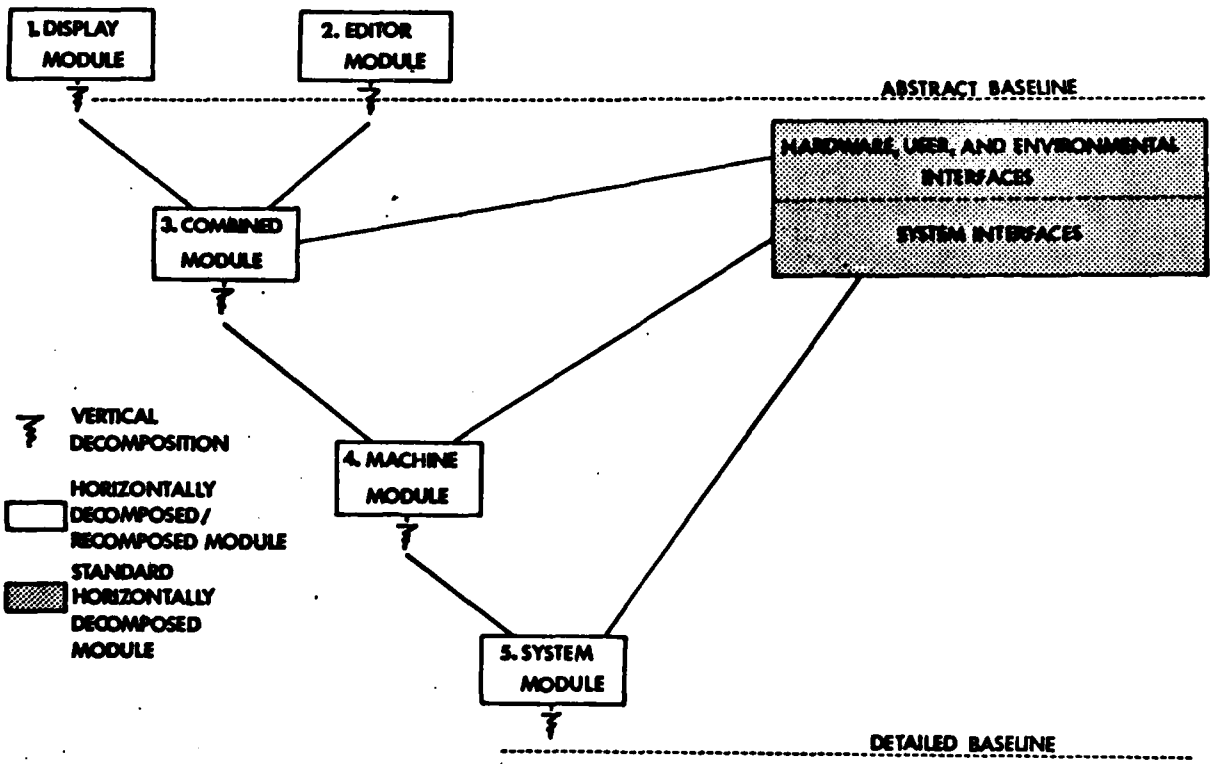
The first decision made in the development of the windowed screen editor system was to separate the filing/viewing functions of the system from the editing and the user command interface functions. This decision represents a logical decomposition of a system which could be difficult to reason about effectively if taken as a whole.

The filing/viewing system will include both the information sharing and the windowing aspects of the system. All decisions about how information

is stored and represented to the user will be made in this component. This will comprise the basis of the system.

The second component, the editor, will deal with the design of a system to update textual documents within the filing system created in the first component. A subset of the *normal* editing functions will be available to the user in the ultimate system design.

The standard components of the machine, the user, and the system interfaces will be recomposed with the two explicitly decomposed modules as required to complete the system.



IV.3 Abstract Design.

The first component of the design has been formulated very abstractly and is presented as Chapter III, A Family of Visible Filing Systems.

The second component of the abstract design is drawn from previous work on the formal definition of an editor.[Sufrin,81b] That abstract editor design has been slightly modified and presented in the style of this methodology in the Specification Library (Appendix A). A good explanation of our editor design appears in [Sufrin,81b] and, therefore, will not be repeated here.

IV.4 Provisional User's Manual

SCREENWRITER
USER'S MANUAL
Contents

1. In the Beginning...

The terminal screen you are using is Screenwriter's way of communicating with you. It will hold 24 lines of information; each line containing as many as 80 characters. The very bottom line is special because Screenwriter will use it to ask you questions, make comments, or indicate when you have done something wrong. Whenever something appears on this line, the bell will ring to draw your attention to it.

The keyboard attached to the screen is your device for communicating commands and answers to Screenwriter. The keyboard is divided into three parts: the character keys, the special command keys, and the directional keys (see fig. 1). The character keys are in the same arrangement as you would find on a typewriter. The special command keys are arranged in three rows and five different areas. The three rows from bottom to top represent increasingly dangerous commands, and some care must be taken in using the top row of commands. In order to cause you to think about your use of these uppermost commands and to prevent an accident, the auxiliary key must be depressed simultaneously with any other key in the top row for the command to be communicated to Screenwriter. The lower two rows of special command keys require only a single key depression for each command. The five areas of special command keys will be discussed in the order they appear on the keyboard. When a key (command) is discussed below, it will be underlined for emphasis.

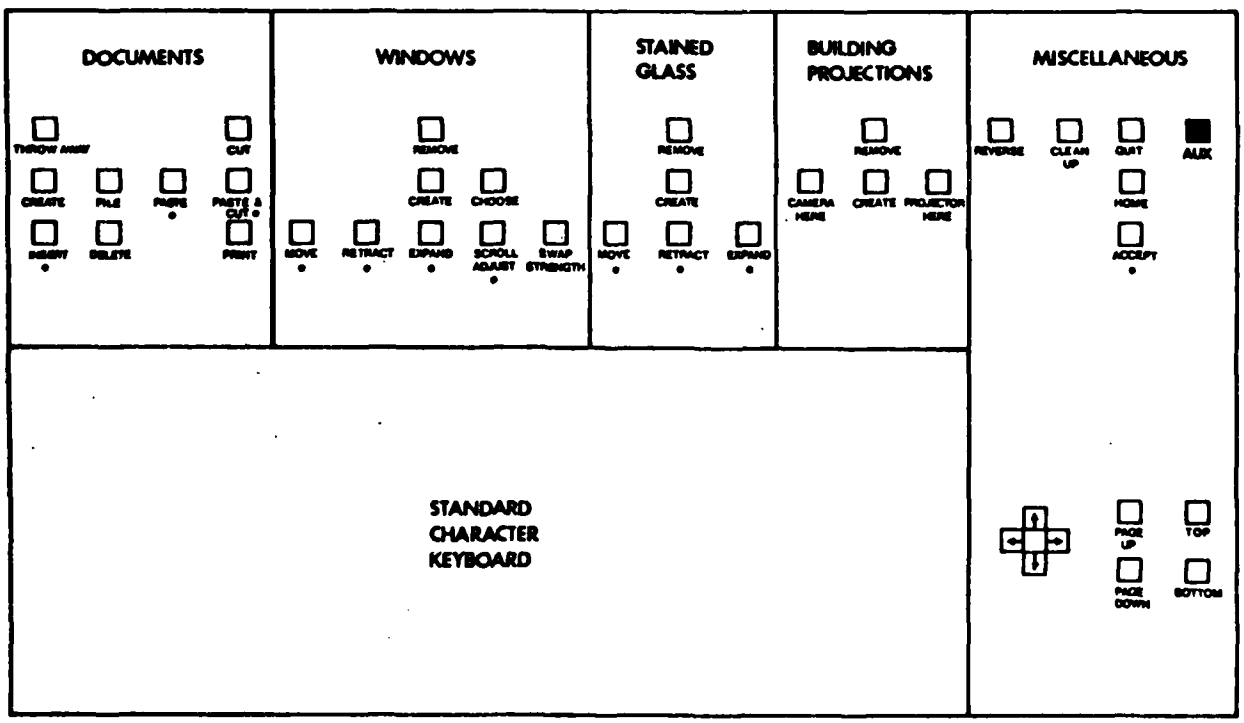


FIG. UM-1 -- The Keyboard

When you call Screenwriter into action it will welcome you with a message on the last line of the screen. The very first time you use it, the upper 23 lines of the screen will be empty to show you that your new office is empty, ready for you to go to work. In subsequent uses, you may start with a picture telling what information had previously been stored with Screenwriter. This picture is called the *home picture* because you will return to it regularly.

The Paper.

For you to create a *document* (eg. letter, memo, report, etc.), the first thing you will require is a blank sheet of paper. The beginning of this blank sheet is easily obtained by depressing the create document key. The paper will be given to you one line at a time exactly as is done on a typewriter. Each line will hold up to 80 characters of typed information, so your sheet of paper will be exactly as wide as the screen. The bright flashing line on the screen is called the *cursor* and is used to point to the current location on your paper. After the create document key is pushed, the cursor will be in the upper left hand corner of the screen and will point to the first position on your new one line sheet of paper. This first line is called the *name line* because it is here that you will type any identifying information about the document you are creating (eg. name, title, draft or final, date, etc.).

The Type.

To put this name information on the first line or to add information to any future line is exactly the same as for a conventional typewriter: simply type the characters that you wish to appear on the page. After each character that you type the cursor will move to the next position on the line. However, the cursor will stop at the end of a line and refuse to go any further, just as your typewriter would not let you beyond your margin. Also like your (electric) typewriter is the manner in which you move to the next line on the paper -- depress the return key. As you would expect, the return key will cause the cursor to move to the beginning of a new blank line, ready for typing. Of course, a completely blank line in your document (eg. double spacing) is the result of two returns in a row.

The Arrows.

At some point in the creation of your document, you are likely to make a "typo", but don't despair! The correction of errors is one of the advantages of Screenwriter over your conventional typewriter. To fix an error in some part of your document, simply use the arrow keys to move the cursor to the location of the error and retype on top of the erroneous part. The new character you type will replace the original character that was there (no erasing is required!). The arrow keys allow you to *roam* around your document making changes as you desire. However, the arrow keys will not push the cursor off the document you are currently creating.

The Insert and Delete Keys.

Two other types of easy corrections are possible with Screenwriter. The delete command will cause the character which is pointed to by the cursor to disappear and the rest of the line will be joined back together as if the deleted character had never been typed. Deleting a blank character after the end of a line will cause the line up to that point and the following line to be joined into a single line (unless doing so would make the resulting line longer than allowed by the margins.) One character is deleted each time the delete key is pushed.

Inserting, as you might expect, does the exact opposite of the delete command. The insert command will switch the machine to the insert mode, thereby causing characters to be inserted before the character denoted by the cursor. Inserting a return in a line will cause the remainder of the line to the right of the new carriage return to be moved to a new line. The word "INSERTING" will appear on the last line of your screen to remind you that any character you type will be inserted rather than replace an existing character. Hence, many characters can be inserted with one push of the insert key. Screenwriter will know the insert command is over when you depress any key other than a character key. (If you will notice, there is an "*" on the insert key on your keyboard. This corresponds to the "*" on the accept key. The accept key is a very safe key and can be used at any time without fear. It will safely end any "*'ed" command which has been started, but will do nothing else.)

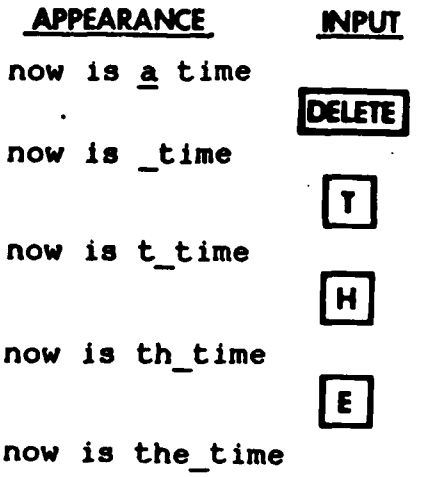


FIG. UM-2 -- Insert and Delete

The Pages.

With standard typewritten documents you are used to the paper being a constant length (say A4) and any document of a greater length requiring more than one page. Screenwriter has a variable size for the length of the piece of paper you write on -- as we noted earlier the paper will grow one line at a time so that it is always just long enough to hold your document. A *screen page* of this document you create will be any 23 consecutive lines that are seen on your terminal screen. (Remember, the 24th line is used for Screenwriter to talk to you.) As you add more than 23 lines to your document, you will notice that lines disappear off the top of the screen as new ones are added to the bottom. This vertical shifting of the information visible on the screen is known as *scrolling*.

Scrolling is rather like what happens when you view a film strip with a projector. The film strip (document) can be almost any length but you will be able to see only one part of it (screen page) at a time. The projector scrolls forward to move to the next part or scrolls back to reveal the previous part. All parts but the one being projected are hidden from view on one of the two film reels. Similarly, all parts of your document will be hidden except the part which has been scrolled onto your terminal screen.

Scrolling to correct your document is accomplished one line at a time by using the up arrow for scrolling up the document, or the down arrow for scrolling down the document. When the cursor reaches the bottom (top) of the screen page, further use of the down (up) arrow will cause the information visible on the screen to shift up (down) one line and the next (previous) line to appear on the bottom (top) of the screen page.

If your document is very long, it may be inconvenient to scroll through it one line at a time in search for a particular line or word. Therefore, it is possible to scroll through your document a full page (23 lines) at a time using the page up or page down command. The page down command, for example, will cause the line previously just off the bottom of the screen page to be moved to the top of the new screen page. Additionally, the beginning or end of a document can be scrolled into view with the top or bottom command.

The Printed Copy.

Once a document has been created and corrected, you will probably want a printed copy of it to send out, give to your boss, etc. Getting a printed copy is as easy as giving the print command. If the cursor is in the document's name line when this key is pushed, then the entire document is printed. If the cursor is anywhere else in the document at the moment print is pressed, then only the current screen page is printed. When Screenwriter is finished printing, it will tell you so on the last line of your screen.

The Filing Cabinet.

If the document you have created is something that you wish to save for reference, or further work, then place it in Screenwriter's *filing cabinet.* Filing a document is done very simply with the file command key. This command files away the document exactly as you currently have it.

The Waste Bin.

If the document you have created is something that you wish to throw away because you have no further use for it, then place it in Screenwriter's ***waste bin.*** Warning: throwing a document away is permanent since Screenwriter will not retrieve from the waste bin. Therefore, the process to get rid of a document is slightly more involved than the process to file a document. First, you must press the throw away key which will cause Screenwriter to ask the question: "DO YOU REALLY WISH TO LOSE THIS DOCUMENT?" If you give any answer to this question except "Y", then the answer and the throw away command will be ignored. The extra difficulty of this command will give you time to consider your decision to throw away

a document you have created and to further protect you from accidental destruction of a document. Completion of the throw away command (i.e. an answer of "Y" to the question) will cause the document to disappear and the screen to return to the *home picture*.

2. Seeing is Believing...

Any number of documents can be created, corrected, printed, filed, or thrown away in the manner described in the preceding chapter. However, these commands do not fully allow you to utilize the flexibility and power of Screenwriter. To tap this power you must understand how to change the picture you see on the screen to suit your requirements. Thus far, you have been able to see one of two pictures on the screen. The first possibility called the *home picture;* was seen immediately upon setting Screenwriter into action or as a result of throwing away a document you've just created. The home picture (which will be explained more fully in a moment) is a very important view of Screenwriter and is probably the starting place for most of the work you will do. The second picture you have been able to see on your screen is one which included part of the single document you were working on at the time. This picture was visible between the create command and the file or throw away command. Remember that while working on a document, Screenwriter would not let you see anything on your screen except that document. Now it is time that you controlled what is visible on your screen.

The Projections.

Imagine for a moment a big white tabletop upon which Screenwriter could display any of the documents you have filed in its filing cabinet. It could do this with one of the projection systems shown below (fig. 3). The camera focuses on the document (or part of it) and that information seen by the camera is projected onto the tabletop. Certainly if you were given enough of these projection systems, you could *lay* the documents in the filing cabinet on the tabletop in any arrangement you desired, and that is exactly what you will do.

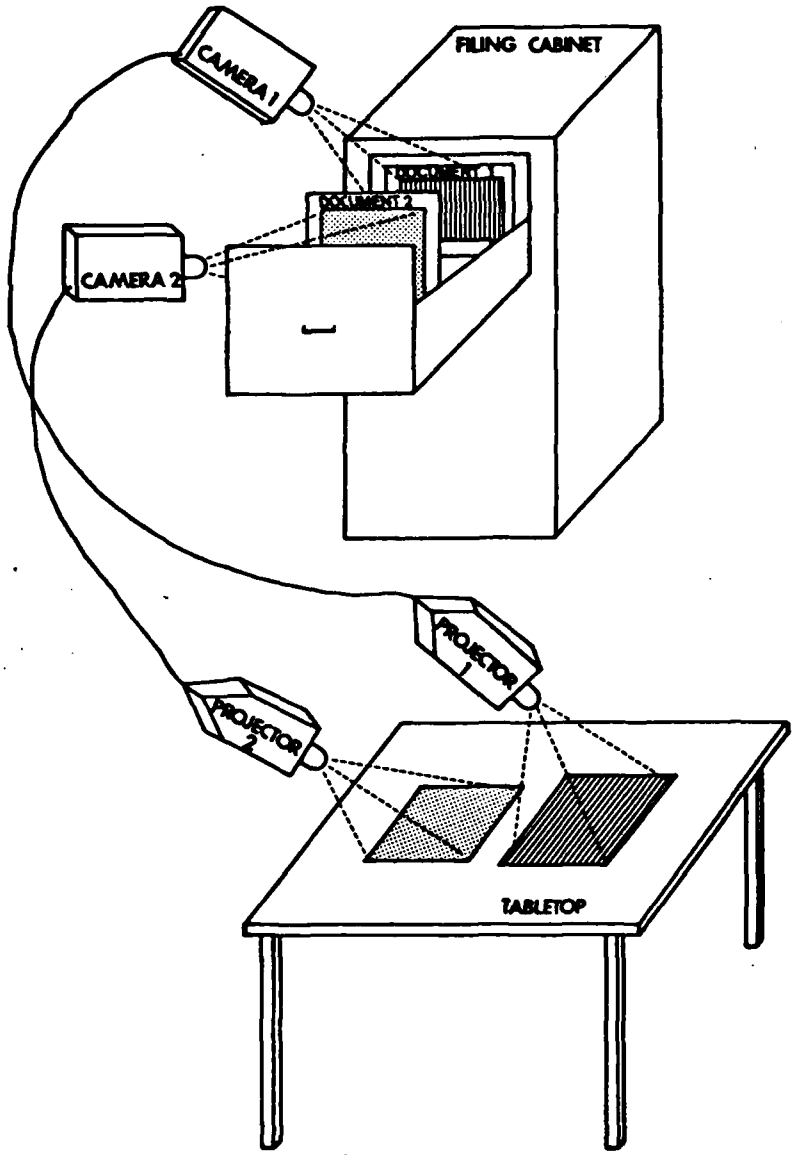


FIG. UM-3 -- A Projection System

Your first comment at this point might be: "I have imagined a very large tabletop but all I see is 23 eighty character lines!" And right you are. The size of your terminal screen limits you to seeing only 23 consecutive lines of the tabletop, but you can control which lines you see by a technique you have already discovered -- scrolling. Remember the arrows, page up, and page down? What those keys actually did was change the area of the tabletop visible on your screen. If you look again at the arrow keys, you will note that there are also two horizontal arrows which allow you to *adjust* left or right to any 80 character segment of tabletop. Of course, the tabletop

has edges, and you would not expect to be able to see beyond the edge of the tabletop. For simplicity, let's call the area of the tabletop that is visible on the screen the *picture area*. As you recall, the picture area when you were creating a new document was called the screen page.

Having resolved that problem you might wonder: "How do I remember and find all of the documents I have created and filed away?" To answer that we return to explain the home picture. Starting in the upper left hand corner of the (rectangular) tabletop you have just imagined, begin to project the name lines of all the documents you created and filed in Screenwriter. Project them onto the tabletop in the order they were created, each below the previous one. Having arranged the name lines of all documents in this manner, insert a bar underneath each name line to represent the number of lines in that document (ie. the longer the bar the longer the document). What you just imagined is the home picture. No matter what part of the tabletop is visible on the screen, it is simple to return to the home picture -- simply push the home key. While you have the home picture on your screen the last line will display the words: "HOME --** DOCUMENTS IN THE FILE" (where ** will be replaced with the document count).

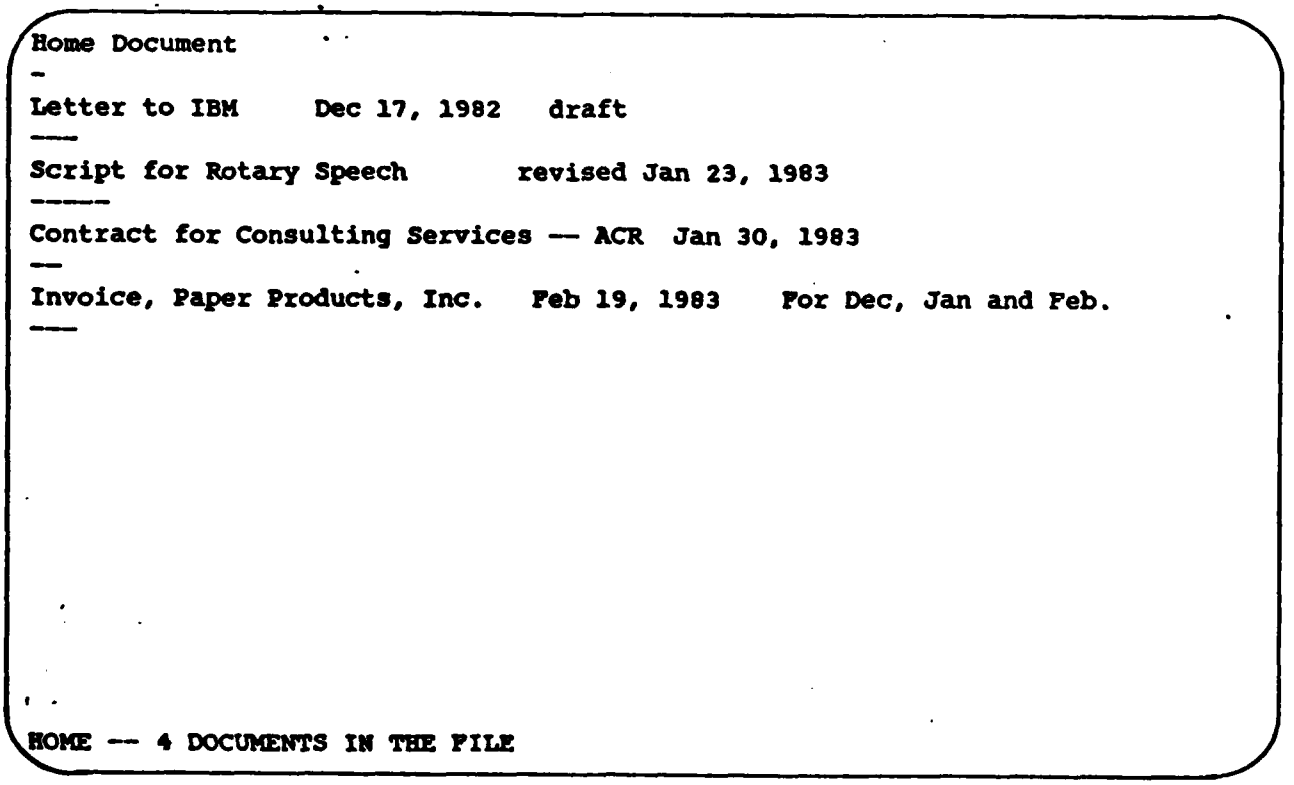


FIG. UM-4 — The Home Picture

Finally, you might ask: "What happens when one of the projections overlaps another projection on my imagined tabletop: is there a double

exposure?" The answer is "no". The projections are of varying strengths, so that in case of overlap only the stronger projection will be seen in the area of overlap. The rule is: the newer projections are stronger than the older projections.

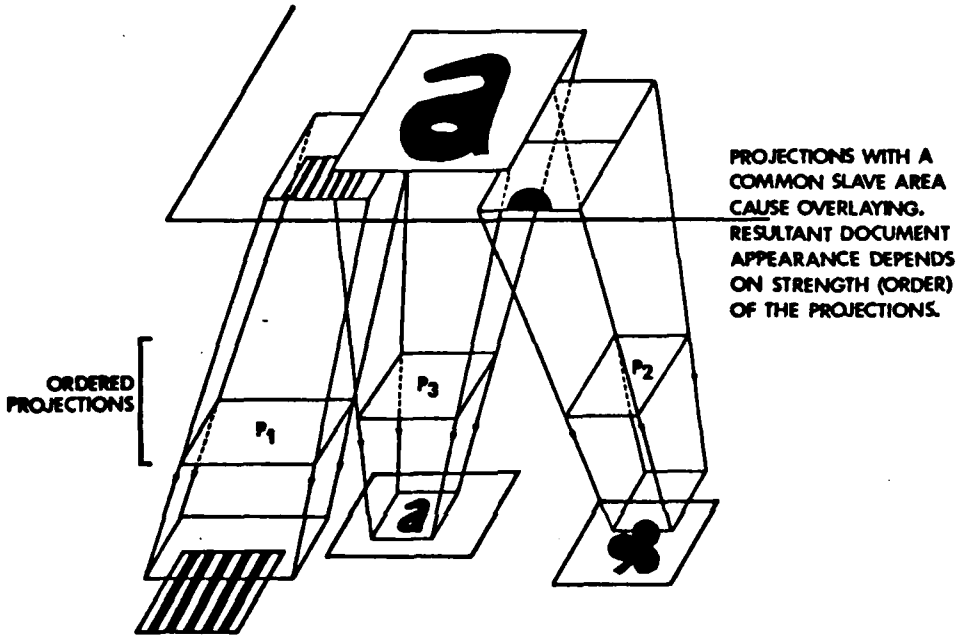


FIG. UM-5 -- Overlapping Projections

The Window

Now that you have a mental picture of Screenwriter busily projecting documents from the filing cabinet to the tabletop, it is time to explain the keys which will cause Screenwriter to create the tabletop arrangement of document parts you desire. Before beginning this explanation, let's remember why it is important to get a specific arrangement of documents on the tabletop. Firstly, we know that the only way we can see documents we have created and filed in the system is to project those documents onto the tabletop and scroll or adjust until that part of the tabletop is visible on the screen. Secondly, we will compare, combine, and change documents; therefore, we want to position the documents on the tabletop so that the relationship among them is appropriate to our needs.

The area of the tabletop covered by a projected document is called a *window*. That is because the area is always rectangular and surrounded

by a dark line which looks like a window frame. At the top of the window is always the name line of the document projected into the window or as much of it as will fit in the width of the window. (see fig. 6) The area of a window is defined by the cursor location on the screen at the time you push the create window key. To position the window correctly, place the upper left corner of the screen exactly at the tabletop spot where you want the upper left corner of the window to be. Now, move the cursor right and down until it is in the spot which is to be the lower right corner of the window. Having done this, the window area is defined by the area from the top of the screen to the line including the cursor and from the left edge of the screen to the column including the cursor. In other words, the line from the upper left corner of the screen picture to the cursor becomes the diagonal of the new window. Note that the largest window you can define in this manner is one the size of the screen. When the window is the size you desire, push the create window key. This key will draw the dark line frame around the area you have defined and *clear* the area inside the frame. After a moment, Screenwriter will take you to the home picture so that you can pick the document you want projected to the window. The bottom line will say "CHOOSE A DOCUMENT TO PROJECT, PLEASE" to remind you of your task.

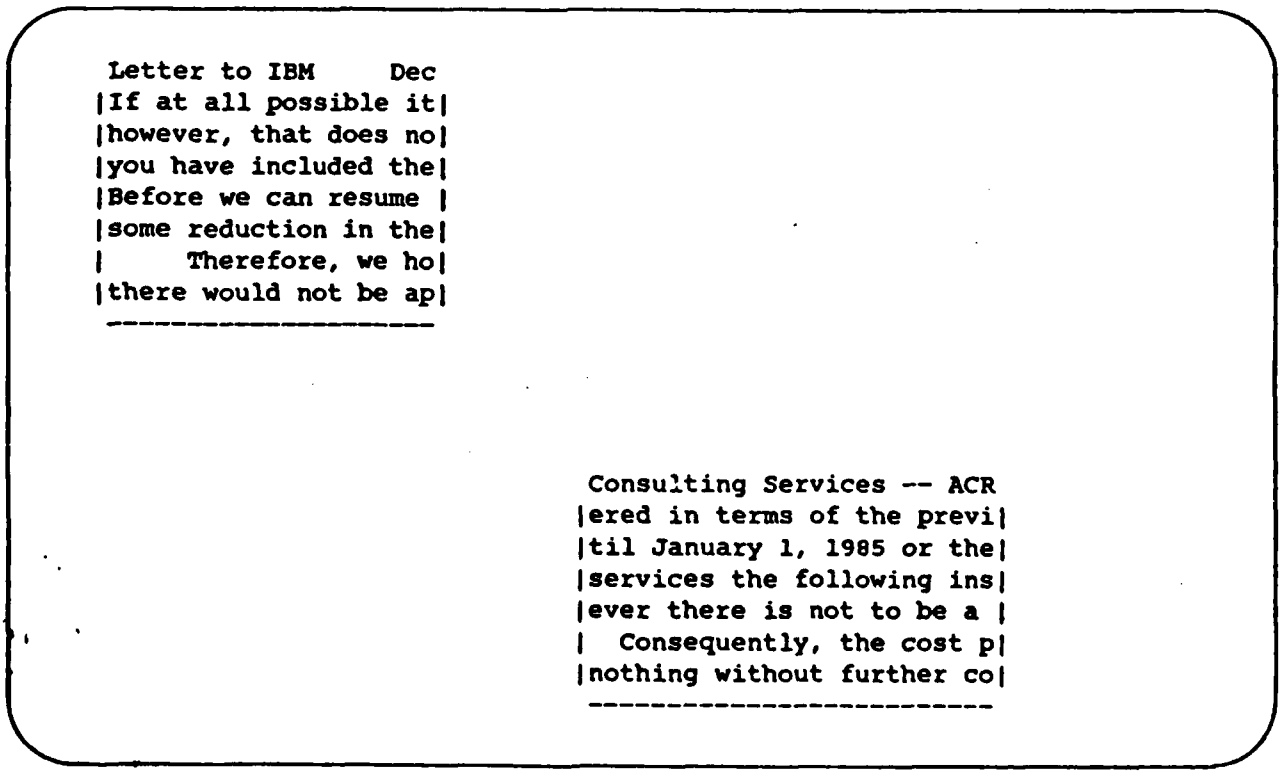


FIG. UM-6 -- Windows

To get the proper document into the window you have created, move the cursor to the appropriate name line. Now, move the cursor across the length bar under the name line until it corresponds to the approximate area of the document that you wish to see in the window. Each position on the length bar represents 10 lines in your document. After the cursor is in the proper place in the name line, depress the choose key. This action will return the screen (and cursor) to the area of the tabletop where your newly defined window can be seen. In the window will be that portion of the document you selected via the length bar.

If you wish to get an entire document from the filing cabinet and project it to the tabletop there is another, simpler method that you may use. The choose key will project the entire document selected by the cursor to the area starting below the home picture if a window has not been defined (ie. you did not depress the create window key immediately prior to the choose command.) Remember that the home key will take you to the name line area of the tabletop anytime you wish to display an entire document in this manner.

The Window Changes.

Now that projections to the tabletop have been created by Screenwriter, you may find that changes in the size, placement or content of the window would be useful. Well, it is quite possible to make any of those changes without difficulty. For all of these commands, first pick the window you wish to change by placing the cursor within it.

Let's see how to change the size of a window. The single rule you must be aware of in adjusting the size of window is that the window can never be larger than the document paper being projected to it. So for example, it is not possible to expand a window beyond 80 characters wide since that is the maximum document paper and projection width. With this rule in mind, push the window expand or window retract key depending on whether you wish to increase or decrease the size of the window. Now, use the arrows to point in the direction you want the window frame to move. With the window expand key the edge of the window frame on the same side as the depressed arrow moves in the direction of the arrow; while using the window retract key, it is the side opposite the depressed arrow which moves in the direction of the arrow. Figure 7 will help you visualize how to use the arrows for these commands. The window cannot be retracted completely: part of the window must always be visible. To indicate to Screenwriter that you have changed the size of the window as much as you wanted, press

the accept key or any other non-arrow key. While you are changing the size of a window, the last line of the screen will say "CHANGE WINDOW SIZE--USE ARROWS".

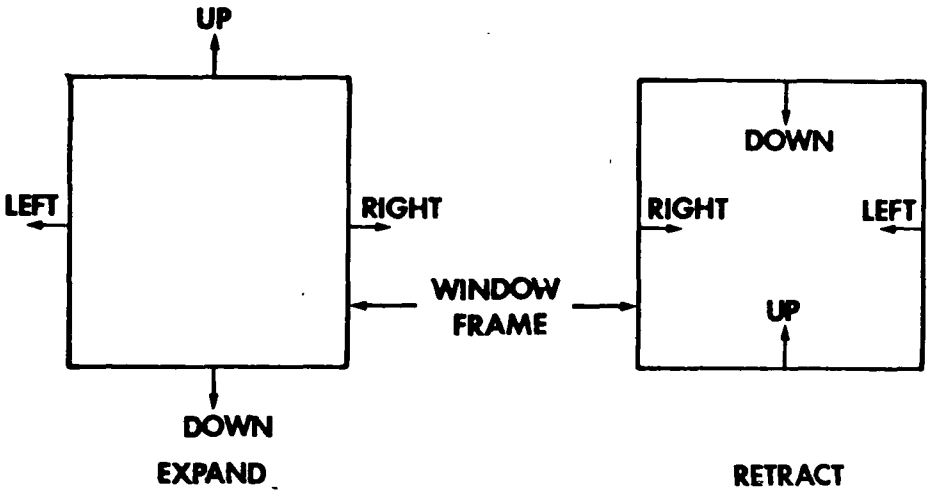


FIG. UM-7 -- Window Changes

Changing the location of a window is similar to changing the size. The rule to remember here is, obviously the window must remain on the area of the tabletop. To move the window: position the cursor, depress move window and use the arrow keys. The window will move around the tabletop in the direction indicated by the arrows that are depressed. To conclude this command, push accept or any other non-arrow key. Again, Screenwriter will remind you by putting "MOVE WINDOW--USE ARROWS" in the last line of the screen.

Finally, the content of the window (i.e. what you see *in* the window) can be easily changed. Just as you can cause Screenwriter to scroll (up or down) and adjust (left or right) the tabletop in the screen, you also can cause the document being projected to a window to scroll or adjust in the window. This will have the result of changing the area of the document which is visible through the window. The requirement, of course, will be to keep the projection camera focused on some part of the document paper; that is, what shows through the window must always be part of the same document. Once again, the cursor position at the instant the scroll document key is pushed will select the window. The arrows, page up, page down, top and bottom will cause the projection camera to move in the direction indicated. As before, the accept or other non-arrow key will end this command. Screenwriter will indicate the command he is performing by placing the words "SCROLL OR ADJUST--USE ARROWS" on the last line on the screen.

The Projection Strength.

As noted earlier, projection strength decreases with age. That is, newer projections will overlay (hide) older projections. There may be times when this causes some difficulty because a more important document gets hidden by a document of less importance. To correct just such a situation there is a command which will swap the projection strengths of two projections on the tabletop. Since the only time that projection strength is important is when hiding occurs, that will be the only time the swap strength command will be effective. To use the command, first move the cursor to point to a spot where one window covers another. Now, press the swap strength key. The result of this command is that the window which was previously hidden in the area of the cursor is visible and the previously visible window is hidden. This swapping changes the strength of both entire windows; so, as a consequence, some other documents may partially disappear or become visible. (See fig. 8.) (Note that if more than one document is hidden, the swap strength command will uncover only the strongest of the hidden documents.)

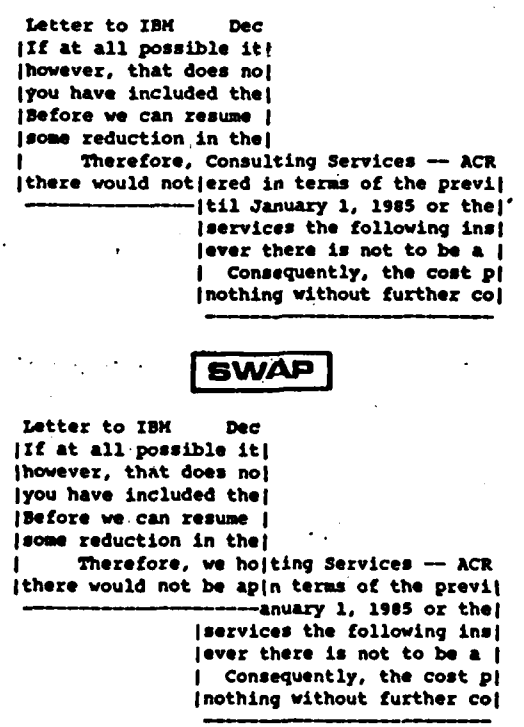
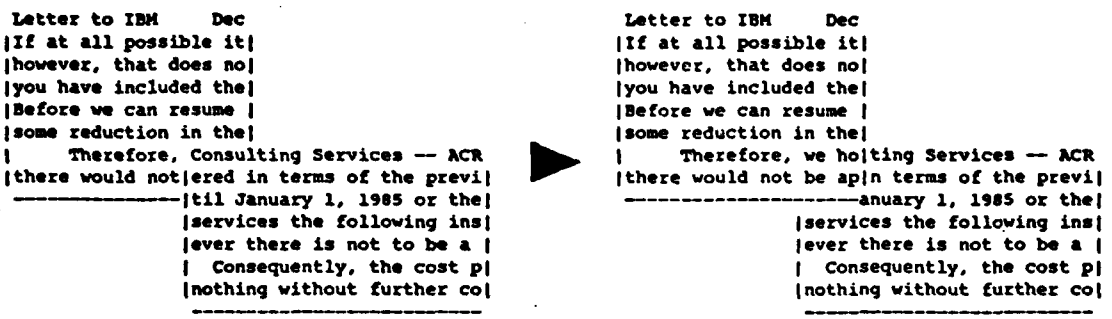


FIG. UM-8 -- Swapping Projection Strength

You may at times have noticed some automatic strength swapping or tabletop scrolling and adjustment. This is how Screenwriter ensures that

part of a window that you need to see to finish a command is visible to you. This is called *disclosure* and was first seen when Screenwriter automatically scrolled up the screen page as you added more lines to the document you were creating. The commands move, retract, and the character or arrow keys are those which can possibly cause disclosure.



OR

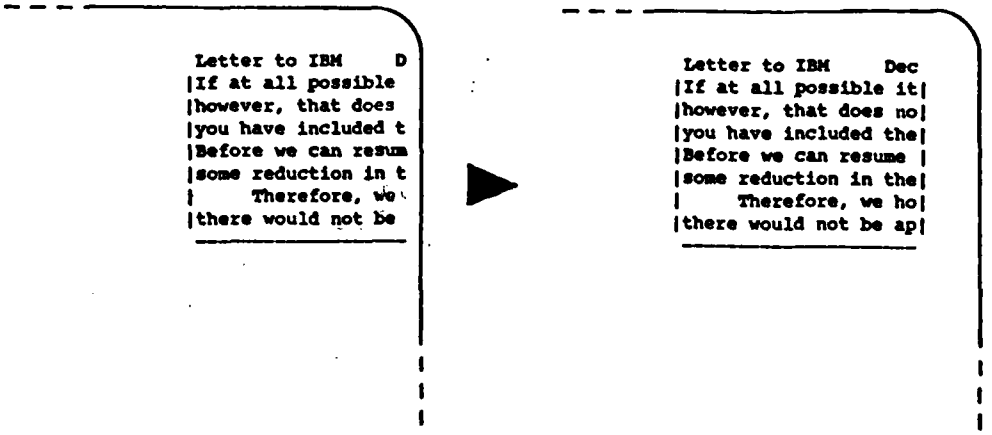


FIG. UM-9 -- Disclosure

The Window Removal.

Once a projected document is complete and it is no longer needed on the screen, it can be easily deleted with the window remove key. The window to be removed must contain the cursor at the time this key is pressed. The result of this action is to change the appearance of the tabletop by totally removing the designated window. The document in the filing cabinet is not removed.

In the first chapter we learned how to use Screenwriter to create, correct, print, throw away, and file documents. In this chapter we learned to retrieve documents which have been filed by projecting them from the filing cabinet to windows on the tabletop. We also now know how to change the window size, location, and content to suit our needs. We can change the relationship between two overlapping windows by swapping their strengths. And finally, we can scroll the tabletop up and down or adjust it left and right in order to display on the screen exactly that portion of the tabletop which is currently of interest to us. With all of this flexibility now at our command, we are prepared to learn how to change any document in the filing cabinet that is visible in a window. (Remember, up to this point we have been able to correct only those documents which were created but not yet filed.)

3. Like a Bridge Over Troubled Water...

It would certainly be painful to have to start completely over on a document if we found an error in it after filing it initially. Sounds like the problem of trying to correct something after removing it from the carriage of your typewriter, doesn't it! Fortunately, Screenwriter is much too flexible to allow that sort of a problem. Much of the correction capability is not new, but is the same capability that you had when initially creating the document (i.e. when the document was still in the typewriter carriage.) That includes the ability to change characters, insert characters and delete characters. In addition, some new, more powerful line insertion and deletion commands can be included in your tool box with only a very brief further explanation of projections.

Before we discuss these document changing commands in greater detail, it is important that you understand the effect such commands will have in general. Earlier, when we were creating and correcting documents, we used a blank sheet of paper which expanded to be as long as necessary to hold one document. Now, we realize that the sheet of paper was simply part of the tabletop, and that corrections were being made to the tabletop itself. With the documents no longer part of the tabletop but instead filed away in the filing cabinet and only their projections visible on the tabletop, what will the corrections actually change? The answer is quite simply that the corrections we will discuss in just a moment will still correct the document.

While that may seem a little strange at first, you will soon realize how useful it is to be able to change a document by working on a picture of it that you have placed in a window on the tabletop. However, one important consequence of this must be emphasized: projections always show what is on the filed document: if that document is changed, then the picture of the document seen in a window will have also changed. Or, put another way, if the same part of a document is projected to two different windows and the document is changed through one of these windows, then the same changes will also be seen in the second window.

The Document Corrections And Printing Revisited.

In the first chapter of this tour of Screenwriter, we encountered some techniques for correcting, removing, or printing a document that was being created. The character keys, insert, delete, print, and throw away were used to accomplish this manipulation of the new (not yet filed) document. These same keys can now be used to manipulate documents in the filing cabinet whose projections are visible in windows on the screen.

For example, typing character keys in a window will change the characters seen in the window, because it changes the document which is being projected to that window. Even the name line can be changed in this way. Just as the character keys would not force the cursor off the document before, neither will they force the cursor out of the window frame now. The size of your window, therefore, will determine the current margins for all changes and additions you make with the character keys. (Note that the window will not affect the line width of text previously created with different margins.) Although not mentioned earlier, a document being created is in a window of its own and, therefore, margins can be set during document creation in this exact same manner (eg. with window retract or expand).

To add characters to existing lines or to create new lines in the projected document can be done as before with the insert command. In this case, the Screenwriter will only allow insertions which do not force the end of the line beyond the edge of the current window frame (margin).

Delete also has the same effect as before. Deleting a carriage return will only work if the two lines being joined will still fit in the margins defined by the window.

The throw away key is just as dangerous as before because it will also cause a document which has been filed to be removed from the file

and placed in the waste bin. To choose a document to be thrown away, place the cursor in its window or its home picture name line and press the throw away key. When this command is complete, Screenwriter will also have eliminated all projections whose cameras or projectors were over the deleted document and its name line will no longer appear on the home picture.

Finally, the print command can now be described in its more general form. When the print key is pushed, the cursor could be in any of three possible positions giving three different printed results. The first possibility is that the cursor is in a name line in the home picture, in which case the whole document represented by the name line will be printed as it exists in the filing cabinet. The second cursor position is in a name line of a projection while not in the home picture. This would cause the contents of that entire window to be printed. The final case is when the cursor is not in a name line. This causes the picture area (ie. that which can be seen on the screen) to be printed.

As you see, these document correction, printing, and removal commands have not changed from the way they were originally introduced, but have merely become more general since they now also work for documents which have been filed in the filing cabinet and projected back to the tabletop.

The Stained Glass.

Before we introduce the new document changing commands which deal with lines of a document rather than single characters, we need some method of pointing to a group of lines that are visible in a projection window. This is done by coloring one of the panes of the window to highlight the part of the document seen through that pane. In many respects this highlighted area will behave like a "window in a window." The methods for creating, expanding, retracting, moving, and deleting a pane of *stained glass* are similar to the equivalent methods used for windows themselves. The stained glass pane must be totally within the frame of a single window just as the window must stay on the tabletop. Perhaps the simplest method of thinking about the stained glass is to imagine a colored filter which has been placed over part of the lens of the projector which is projecting into the window frame.

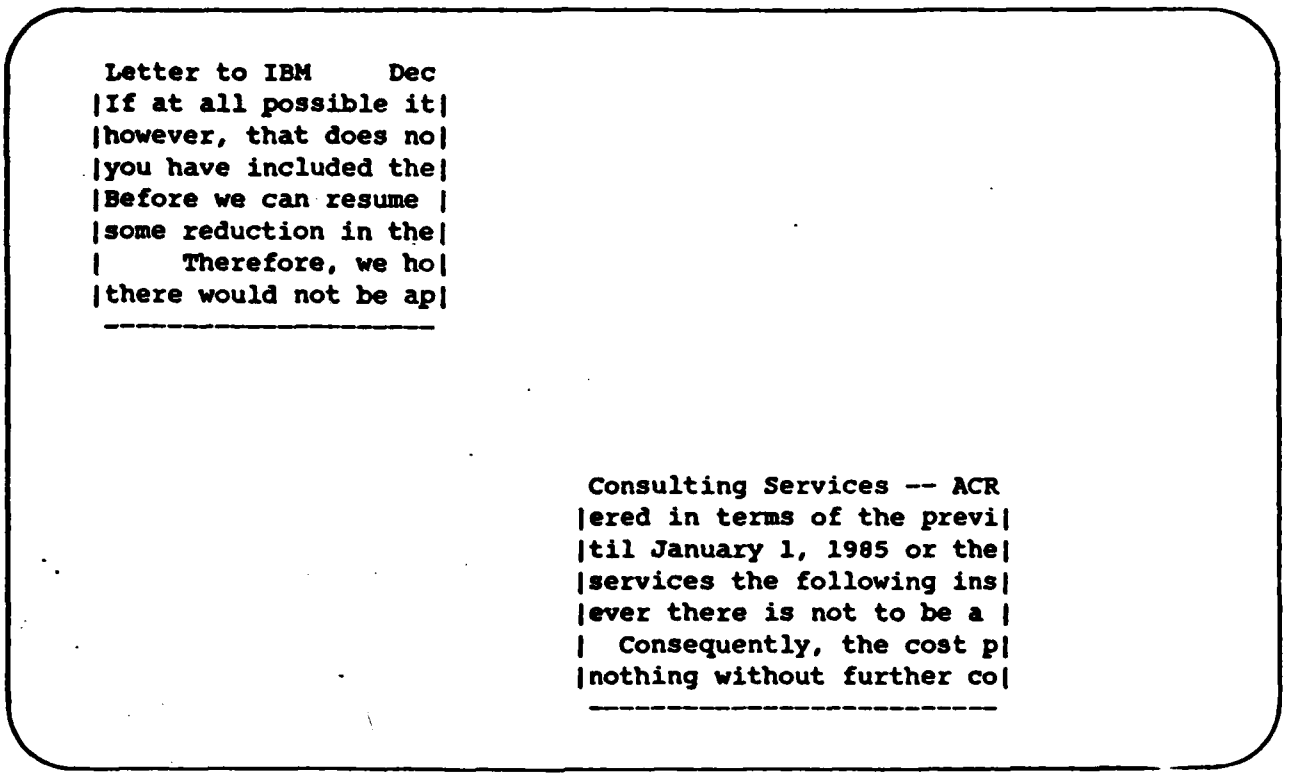


FIG. UM-10 -- The Stained Glass

To create a stained glass window pane, scroll the tabletop until no area of the window which is above or left of the intended pane is visible on the screen. Then, move the cursor until it is in the lower right corner of the intended pane. (Note how similar this is to the way you identified a window area -- the only difference is that the pane must be totally within the window area.) Finally, press the stained glass create key and watch the area you have indentified turn to reverse video (black on white) like the name line. If instead of a small stained glass pane you want the entire window to be stained glass, then place the cursor in the name line shown in the window and press the stained glass create key.

Stained glass expand, retract, move, and remove work as described for the window commands of the same name (once again remembering that the stained glass pane must stay in the window.)

The Scissors and Glue.

Have you ever wanted to snip a few lines or a paragraph out of a document or get out your glue and add a new paragraph to your document? Screenwriter will help you do both of these document updating tasks. For

all of the document updating commands (cut, paste, and paste&cut) you will create a stained glass pane to indicate the lines that you wish to move or remove. The window will be selected by the cursor position at the time the command key is depressed. Any lines in the window which are even partially colored by the stained glass will be included in the area to cut or pasted. A strong word of warning should be issued here: know what part of your window is in stained glass before you use one of these commands -- the entire pane of the stained glass will be used as a line selection device regardless of whether or not you can currently see the entire pane.

The cut key will remove from a document all lines which are partially covered by the stained glass in the window selected by the cursor. A document can not be completely removed in this manner because the name line can not be snipped away. Of course, any gap created in the document by the cut command is immediately closed by Screenwriter who will join the two parts back together.

The paste command requires two keys. First, the paste key will cause Screenwriter to copy all the lines which are partially covered by the stained glass in the window selected by the cursor. These copied lines will then be inserted in the place you choose. To choose a receiving location, position the cursor in the line of the receiving document immediately above where the new lines are to be inserted. The accept command will complete the action by pasting the lines into the selected location. After you depress the paste key, Screenwriter will remind you that it is waiting to paste in the copied lines with "PASTING--POSITION WITH ARROWS, END WITH ACCEPT" on the last line of your screen.

The paste&cut key does exactly the same as the paste followed by the cut. The effect is to transfer lines of a document rather than just copy them as is done with the paste command. Like the paste command, paste&cut is terminated with the accept command.

This chapter has taken us through a review of the keys which are useful for document correction; like the character keys, insert, delete, and throw away. Then we found the secret of highlighting part of a window by putting in stained glass. With this method of selecting lines from a document, we found some stronger document updating techniques in cut, paste, and paste&cut. However, this does not complete the tool kit that Screenwriter gives you for handling documents. The next chapter outlines the final tool.

4. Deja Vu...

Screenwriter as you currently know it, is certainly a powerful warrior in the "paper war". But, the power does not end here; indeed, the best is yet to come. To study the final capability available in Screenwriter for dealing with documents we need to solve the riddle: "How can part of one document actually be part of a different document?"

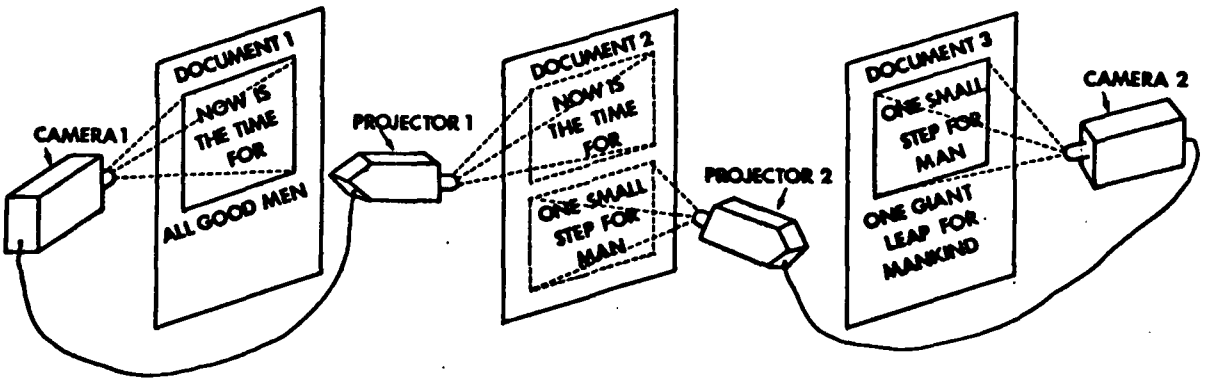
The Projections -- Yet Again.

The answer to the above riddle is similar to an answer we used to solve another problem earlier. Projecting text from one document to another will cause the shared information to exist in both documents. A quick review of projections will show that there is an even greater advantage to projecting text from one document to another. (After all, paste already allows us to copy text from one document to another.)

Recall that a projection involves a camera which *sees* part of one document and transmits what it sees to a projector which projects that same picture onto a document. (If you have come to the conclusion at this point that the tabletop is just a large document, you are absolutely right.) Both the area seen and the area projected to must be on an existing area of a document. The strength of a projection depends on how recently it was created (i.e. new is strong, old is weak) and the relative projection strengths determine what is visible in an area where two or more projections overlap. The final attribute of projections to remember is that any change made to the area of the document seen by the camera will also be seen on the document that is projected onto. That is, by changing the original you also change all projected images of that original. It is this last attribute that makes projections such a useful tool in the building of documents. As an example, consider the possibilities where an address or telephone number changes. If you projected the address (or telephone number) rather than copying it whenever it was required on a document, then a single change to the original address or telephone number would correct all occurrences of it in your file of documents.

Two commands, using four keys, will allow you to utilize the power of *document building* projections. Document building projections are projections which do not involve the tabletop, but rather, involve only documents in the filing cabinet. (Of course, you already know about the commands which create

and control *viewing* projections, that is projections which cause documents to appear in windows on the tabletop.) The first command is document building projection create; it requires that both ends of the projection have been defined. The area to be placed under the projection camera is designated by the stained glass pane in the window pointed to by the cursor when the camera here key is depressed. The projections will be put in the area of the appropriate size whose upper left corner is defined by the cursor position at the moment the projector here key is pushed. Once the camera and projector position have been determined in this manner, the create key will create the projection and the effect will be immediately seen on the document under the projector. Whereas the paste command will add a number of whole lines to a document, creating a projection covers part of a document which already exists and is not at all confined to whole lines. After the camera or projector positions have been selected, either can be changed without disturbing the other in order to create a different projection. The only requirement is the areas selected must still exist in the documents when the create key is activated.



DOCUMENT 2 IS BUILT FROM INFORMATION "SHARED" FROM OTHER DOCUMENTS

FIG. UM-11 — Document Building Projections

Perhaps some examples are required here. Suppose one of your documents is an address list of clients and that another is a form letter to be printed and sent to each client. In this case you would certainly want the area seen by the camera to change to a new address before each projection was created. However, the area of the form letter (inside address

area) to receive the projections would not change. Suppose, on the other hand, that you had created a revised paragraph to replace a paragraph currently in a number of contracts filed by Screenwriter. In this case the camera would stay positioned over the revised paragraph while you would position the projectors over the approprate part of the various contracts.

The final command for use on document building projections is remove. To remove a projection, place the cursor in the area being projected onto and push remove. In the area pointed to by the cursor you will immediately see whatever was on the document but had been hidden by the projection just deleted.

It is important to note here that the commands insert, delete, paste, cut, and paste&cut will not work on lines which include text that has been projected onto them in the manner just described. The reason for this is that the text being projected can not be moved or removed except by changing the projection or by changing the text under the camera. The text being projected can not be altered by changing the area onto which it is projected. The throw away command will cause Screenwriter to remove any building projections from or onto the document to be deleted.

Before leaving document building projections behind, let's look at a couple of questions that may have occurred to you as you read this last chapter. For example, you might well be thinking: "When do I use a viewing projection instead of a document building projection?" The answer is, anytime you wish to see a document part on the screen or wish to make changes (including new document building projections) to a document, then the document must first be projected to the tabletop with a viewing projection. If you wish to have the exact same text in two different document areas so that changes to one will be reflected in the other, then create a document building projection.

You may also have wondered: "Why can document building projections not be moved, or changed in size or content like viewing projections?" The answer to this is the different intent of the two types of projections. Document building projections are not likely to change while viewing projections are almost certain to change. If for some reason a document building projection must be changed, then it should be removed and recreated as required.

Another question that may have come to mind is: "If I make a change to a document (say replace one character by another), then what affect does that change have on other documents in the filing cabinet and the tabletop?"

This is an easy but important question to answer. Any projection (viewing or document building) whose projection camera sees the area of the document which has been changed will project that change (to the tabletop or the receiving document.) In the case of the document building projection, the document receiving the projection will be changed and the whole process is repeated if the changed area of this document is under a projection camera. Hence, with the proper projections, it is possible for a single change to be reflected in any number of documents and in any number of windows on the tabletop.

5. To err is human...

It is quite likely that at some time you are going to accidently press the wrong key or change your mind after you see the result of the action you have just taken. Screenwriter has provided for such eventualities by providing a single reverse key to undo what you have just done. Reverse is limited to undoing the immediately preceding command. Therefore, as soon as a new reversible key is activated, it becomes too late to reverse the effects of a previous key. There are, however, some keys which, because of their simplicity to correct by other means, are ignored by the reverse key. These keys are:

1. all character keys and return
2. the arrow keys, page up, page down, top, and bottom
3. accept
4. delete
5. print
6. reverse
7. quit

That is, any one of the other special Screenwriter command keys can be reversed if it is the most recently activated of these keys. Reversing the home key returns the screen to its position at the moment when the home key was depressed. Multiple key commands (eg. those using arrows to give direction) are reversed completely -- thereby reversing the effects of the entire command.

It should also be obvious that reverse is not the only way to undo actions you have commanded Screenwriter to perform. If you look once again to fig. 1 which shows the keyboard arrangement, you will note that the lowest level of special Screenwriter command keys includes only keys which cause movement, size change and character insert/delete (and printing). Any of these commands (except print) can be easily undone with one of the commands from the same level and often by using the same command in the opposite direction. Of course, this type of change can be done at any time and is, therefore, more general than reverse. The middle level of special command keys are mainly the *create* keys and can easily be undone with the upper level *destruction* keys. The destruction keys, of course, are much more difficult to undo than the other keys. Often, the alternative to the reverse key for undoing the effects of a destruction key is the complete recreation of the object (eg. document, window, stained glass pane, or projection) which was destroyed. The top level of special command keys, consequently, must be used with caution. This is why each one requires the auxiliary key to also be depressed, and why Screenwriter may question your desires even further. Note that the only way to reverse the effect of the quit key is to reactivate Screenwriter.

We always hope never to made a mistake and push the wrong key, but if it happens, Screenwriter will usually help you correct the error if you simply tell it to reverse the key.

6. And to all a good night...

The capabilities of Screenwriter have now all been explained and so all that remains is to clean up the tabletop and ensure that the documents are safe and secure in the filing cabinet. The clean up key will cause Screenwriter to do this for you. Since this is a destructive command, Screenwriter will ask you permission to carry out the command by asking: "SHALL I CLEAR THE TABLETOP?" If you reply by depressing the "Y" key then Screenwriter will remove all windows from the tabletop and send you to the home picture.

The quit command stores away all projections and documents so that the next time you activate Screenwriter you will be able to begin work where you left off.

Hopefully you now understand what Screenwriter does for you in the creation, display, and storage of documents. Each time you push a key, try to visualize how it is setting to work on the documents, filing cabinet, projections, or tabletop in order to carry out your command. This approach will help you understand what to expect each key to do for you so that you can more effectively employ Screenwriter.

CHAPTER V

The Detailed Baseline

Included here from the detailed baseline documentation is the final portion of the formal design specification and the system implementation plan. A partial implementation based on this specification and implementation plan has been accomplished, but will not appear in this volume.

V.1 Detailed Design.

This design is a continuation of the filer/viewing system design specification presented in Chapter IV. The other decomposed components of the system will be recomposed with this component in the later levels of the design.

FILER4

Basis:

Prior: FILER3

Forward: FILING CABINET4.1 / PROJSET4.2 / DISPLAY4.3

Comments:

This level decreases the specification to include only the family of systems with rectangular and finite components such as projection slaves and masters, areas of filter application, documents, and the display (screen).

The decision to require these components be rectangular in area is not as restrictive as might first be imagined. Nearly all documents (from *margin* to *margin*) and screens are, by their very nature, rectangular. With this in mind, consider the types of information in this mainly textual system which the user is likely to project or filter. He might project a speck, a *word*, a *line*, a *paragraph*, a *page*, a document, or some multiple of one of these. Obviously, in our rectangular coordinate system a speck (and hence also a *word*) is rectangular. Now, if we consider a document as rectangular from *margin* to *margin*, then the rectangular lines must be constant length within a document. *Paragraphs* and *pages* (each a series of consecutive *lines*) would also be rectangular. Any information of a less *normal* shape which the user might wish to project or filter could easily be described as a combination of these basic rectangular objects.

Additionally, it makes sense to require that there be an even slightly more rigid connection between the shape of a slave and its master than a shared rectangularity. Since it is unlikely that a user would wish to respell a *word* (ie. mix letters of the *word*) as it is being projected, for example, the relative position of each point in a slave should be the same as the corresponding master point relative position in the master. This means that the projection is not only rectangular on both ends, but that the two rectangles are equivalent in size and shape, with projections going between corresponding positions within the rectangular areas. This same *equivalence* relationship should also be true between the screen area and the area of document seen on the screen.

The requirement that the parts of the system be finite is not significantly restrictive to the system since we are using a discrete (integer) coordinate space.

FILING CABINET4.1

Basis:

Prior: FILING CABINET1.1 / FILING CABINET3.1

Comments:

The documents in the filing cabinet are rectangular in shape and must be finite in size.

Requirements / Design Decisions:

[[1]] All documents must be finite and rectangular (that is, all lines are a constant length).

[[2]] For simplicity, all documents must start in coordinate (1,1). This in no way limits the document but does facilitate the system specification.

Auxiliary Definitions:

We can now define the meaning of *being rectangular*. We know that a rectangle is always uniquely defined by its diagonal. Therefore, we define a pair of functions to find the *smallest* and the *biggest* of a finite set of coordinates.

a.

smallest,
biggest : $\mathbb{F}_1$ (coordinate) → coordinate
smallest (S) ≙ μ(c : S | (∀a : S) (c ≤ a))
biggest (S) ≙ μ(c : S | (∀a : S) (c ≥ a))

The smallest coordinate is the *leftmost* coordinate in the *top row* of coordinates. The biggest coordinate is the *rightmost* coordinate in the *bottom row* of coordinates. These definitions are consistent with our lexicographical orderings ≤ and ≥.

Now, determine which coordinates a rectangle defined by the smallest and the biggest coordinate should include.

b.

area : coordinate × coordinate → $\mathbb{P}$ (coordinate)

area ≙ (λa,b)

(x(a)..x(b)) × (y(a)..y(b))

This defines the set of coordinates which is rectangular based on given smallest and biggest coordinates.

c.

is-rectangle : $\mathbb{P}$ (coordinate) → boolean

is-rectangle ≙ (λC)

C ≠ ∅ ⇒ area (smallest(C),biggest(C)) = C

An empty set of coordinates is rectangular by definition. Any other set of coordinates is rectangular if it includes only the coordinates in the rectangular area defined by the biggest and smallest (ie. the diagonal) of the given set.

State Component Definition:

FILING CABINET4.1

FILING CABINET1.1

(∀x : dom (documents))

[[1]] (is-rectangle (dom (documents (x))) ∧

[[1]] dom (documents (x)) ∈ $\mathbb{F}$ (coordinate) ∧

[[2]] smallest (documents (x)) = (1,1))

This restriction of FILING CABINET requires that all documents be rectangular and finite.

PROJSET4.2

Basis:

Prior: PROJSET3.2 / FILING CABINET4.1 (Auxiliary Definitions)

Comments:

The slave and master of all projections must be rectangular, finite, *equivalent* areas. The area transformed by any non-identity filter in a projection must also be rectangular. These are reasonable restrictions because we commonly work with only rectangular areas of text. This specification restricts the PROJSET2 appropriately to enforce these requirements.

Requirements / Design Decisions:

[[1]] All projection slaves and masters are finite, rectangular areas.

[[2]] The slave and master of any projection are equivalent areas in size and shape, with the projection defining corresponding positions within these areas. Besides ruling out scaling and skewing, this eliminates any arbitrary information jumbling.

[[3]] Any area of non-identity filter application within a projection slave area is finite and rectangular.

Auxiliary Definitions:

a.

$$\text{strip} : \mathbb{P}(\text{COORD}) \rightarrow \mathbb{P}(\text{coordinate})$$

$$\text{strip} \triangleq (\lambda S)\ \{\text{coord}(s) \mid s \in S\}$$

This reduces a set of COORD to a set of coordinates by removing the docname from each COORD.

b.

```
is-eqshape : projection → boolean
is-eqshape ≜ (λP)
             (∃n)
             P = (λc) (c + n)
```

This constant defines the concept of equivalent areas of size and shape for the slave and master of a projection. Less abstractly we could say that for any point in the slave, the corresponding point in the master must be equal to the difference between the slave coordinate corresponding to the smallest point in the slave, and the smallest point in the slave minus the given slave point. In other words, the smallest slave point and its corresponding master point are used as the origins of the two areas and all other coordinate positions are computed relative to these.

State Component Definition:

PROJSET4.2

PROJSET3.2

```
(∀x : dom (projlist))
[[1]]  (is-rectangle (strip (dom (projlist (x)))) ∧
            dom (projlist (x)) ∈ 𝔽(COORD) ∧
[[3]]   is-rectangle (strip (dom (filterlist (x)
                        / 𝕀(speck))))
        ∧
[[2]]   is-eqshape (projlist (x)))
```

PROJSET3.2 is restricted as required. The fact that the domain of a projection is finite implies that the range of that projection and its associated area of filter application are also finite. If the slave of a projection is rectangular and the master is the same shape, then it is obvious that the master is also rectangular.

DISPLAY4.3

Basis:

Prior: DISPLAY3.3 / FILING CABINET4.2 (Auxiliary Definitions)

Comments:

This restricts the screen to a rectangular area which displays a rectangular area of the chosen document. For textual displays, this is a normal restriction.

Requirements / Design Decisions:

[[1]] The area which is displayed on the screen must be rectangular and finite.

State Component Definition:

DISPLAY4.3

DISPLAY3.3

[[1]] is-rectangle (strip (seearea)) ∧
seearea ∈ $\mathbb{F}$ (COORD)

FILER4

State Definition:

$$\begin{aligned} \text{FILER4} \triangleq\ & \text{FILER3} \\ & (\text{FILING CABINET1.1} \rightarrowtail \text{FILING CABINET4.1}; \\ & \quad \text{PROJSET3.2} \rightarrowtail \text{PROJSET4.2}; \\ & \quad \text{DISPLAY3.3} \rightarrowtail \text{DISPLAY4.3}) \end{aligned}$$

This new state space is FILER3 with the accumulated component restrictions added.

FILER4A

Basis:

Prior: FILER4

Forward: FILING CABINET4.1A / PROJSET4.2A / DISPLAY4.3A

Comments:

This first refinement introduces arrays as a representation of tables and functions. This representation eases the specification of *rectangular* while bringing the specification closer to a high level language implementation.

Additionally, it introduces a different and more convenient method of defining a rectangular area of coordinates as required for slaves, masters, ares of filter application, and the area visible to the user. A finite rectangular plane can be fully described by its starting point (eg. upper left corner) and its size (eg. the length of its diagonal.) Although this is not the only possible representation for a rectangular plane, it has the desirable feature that compositional operations (see FILER1) can be specified with addition and subtraction of coordinate pairs.

FILING CABINET4.1A

Basis:

Library: ARRAY

Prior: FILING CABINET4.1

Comments:

The use of a table (array) is a common and well understood method for storing a number of individual pieces of information (in this case specks). It is also useful because it can be easily defined in a finite rectangular shape to accommodate previous design decisions. Of course, high level programming languages allow the declaration of arrays so this step puts the specification slightly closer to the eventual implementation.

Auxiliary Definitions:

a.

DOCUMENTCLASS

```
table  : ARRAY [ speck ]
rows   : ℕ
cols   : ℕ
```

```
rows = length (table)

(∀x : 1..rows) (length (table (x)) = cols)
```

We create the new description of a rectangular document using an `array`. This `array` will be a `table` of specks which are the content of the document. The size of the table is defined by `rows` and `cols` which represent the number of rows and columns in the table. Of course, this makes the shape of the table rectangular and finite. The `table` will correspond to the `extent` of the document as we previously defined it; however, there is no restriction from allowing part of the background paper (ie. the margins of the document) to be included in the `extent` if we choose to define it that way.

b.

```
retv-surface : DOCUMENTCLASS → surface
retv-surface ≙ (λD)
                  (λc | (1,1) ≥ c ≥ (cols,rows))
                  table (y (c)) (x (c))
```

Given a document description and a coordinate in the extent of the document, this function will return the speck which appears at that coordinate. The proper speck is generated by entering the table at the row and column defined by the coordinate. This function causes the table to appear as if it were a surface.

State Component Definition:

```
FILING CABINET4.1A
      FILING CABINET4.1
      docdata : docname ⇸ DOCUMENTCLASS
```

```
      dom (documents) = dom (docdata)

      (∀x : dom (docdata))
            (biggest (dom (documents (x))) = (cols,rows))

      (∀x : dom (docdata))
            (documents (x) = retv-surface (docdata (x))
```

For each document, there is an `table` which is accessible by the document's name. The `tables` are exactly the same size as the extent of the corresponding document so that a one to one translation between the document and the `table` representing it is easily done. Remember, to this point the only design decision taken about documents is that they must be finite and rectangular -- conditions which are obviously enforced by this refinement.

*** Notice: in future usage, since `rows` and `cols` are derived components, we will use the abbreviation:

```
      docdata (x)
```

to represent:

```
      table (docdata (x))
```

PROJSET4.2A

Basis:

Library: FUNCTION

Prior: PROJSET4.2

Comments:

When two related areas of tables are to be equivalent in size and shape, the easiest description possible of the projection relating those areas is two starting spots and a size/shape description. And, when the areas are rectangular, the size/shape description is simplified to a pair of row and column size values. Similarly, a rectangular filter application area can easily be defined. The actual filter is a total function which is also described by an `array`.

Auxiliary Definitions:

a.

PROJCLASS

```
slavespot     : COORD
masterspot    : COORD
projsize      : coordinate
startpane     : coordinate
panesize      : coordinate
f             : FUNCTION [ speck ]
```

```
aread(startpane,panesize) ⊆
      aread(coord (slavespot),projsize)

projsize > (0,0)

panesize > (0,0)
```

Any projection between rectangular areas on rectangular documents can be described by two starting points (`smallest` points of the slave and

master) and a size denoted by the number of rows and columns included in the projected area. The single rectangular filter application area is described by a starting point (smallest point) and a size. Note that the first predicate requires the area of the filter application to be completely on the slave area. The filter to apply in the specified pane (like a window pane) area is represented in the form of an array with two rows -- one row for the domain and the other for the range -- with the column numbers acting as a link between the appropriate members of each row.

b.

```
aread : coordinate × coordinate → P(coordinate)
aread (a,s) = area ( a, a + s - (1,1) )
```

This simple function allows the creation of a rectangular area from a smallest point and a size rather than a smallest and a biggest point.

c.

```
retv-projection : PROJCLASS → projection
retv-projection ≡
        (λS)
        (π COORD | dn = dn (slavespot (S)) ∧ coord ∈ w)
           dn'     = dn (masterspot (S))
           coord'  = coord (masterspot (S)) + coord -
                             coord (slavespot (S))

where
        w = aread (coord (slavespot (S)), projsize (S))
```

Given a refined projection description, any coordinate in the slave area is translated to a coordinate in the master area by adding the vector from the slave starting point and the given coordinate, to the master starting point.

d.

```
retv-filter-apply : PROJCLASS → filter-apply
retv-filter-apply ≜ (λS)
      ((COORD → {∅ (speck)}) ↑ w) @
                ((COORD → {retv-filter (f (S))}) ↑ z)

where
w = {c : COORD | dn (c) = dn (slavespot (S)) ∧ coord (c)
                          ∈ ww}
z = {d : COORD | dn (d) = dn (slavespot (S)) ∧ coord (d)
                          ∈ zz}
ww = aread (coord (slavespot (S), projsize (S))
zz = aread (startpane (S), panesize(S))

  and     retv-filter : FUNCTION → filter
          retv-filter ≜ (λF)
                        (λs : speck)
                        F (2) (F (1)^-1 (s))
```

Given a refined `filter-apply` description, a coordinate in the filter application area of the slave will return the filter represented by the **FUNCTION**, while a coordinate in the slave area but outside the area of filter application will generate the identity function.

State Component Definition:

PROJSET4.2A

PROJSET4.2

projdata : SEQ [PROJCLASS]

---

dom (projdata) = dom (projlist)

(∀x : dom (projdata))
(projlist (x) = retv-projection (projdata (x)) ∧
(filterlist (x) = retv-filter-apply (projdata (x)))

It is obvious here that the previous design decisions concerning the size and shape of the slaves, masters, and areas of filter application are inherent in this new representation. Indeed, so complete and accurate is this representation of PROJSET that all previously defined restrictions on the state are implicit in this new state representation.

DISPLAY4.3A

Basis:

Prior: DISPLAY4.3 / PROJSET4.2A (Auxiliary Definitions)

Comments:

As done in the FILING CABINET and PROJSET, the rectangular area in the DISPLAY is decribed by a starting point and a size value.

Auxiliary Definitions:

a.

DISPLAYCLASS

```
screenstart    : COORD
screensize     : coordinate
```

```
screensize > (0,0)
```

The rectangular screen is defined by a starting point (smallest point) and a size denoting the number of columns and rows on the screen.

b.

$$\text{areadC} : \text{COORD} \times \text{coordinate} \rightarrow \mathbb{P}(\text{COORD})$$

$$\text{areadC}(C,a) = \{c : \text{COORD} \mid \text{dn}(c) = \text{dn}(C) \wedge \text{coord}(c) \in \text{aread}(\text{coord}(C),a)\}$$

This function creates a rectangular area on a specific document from the smallest point of the area on the document and the size of the intended area.

c.

retv-display : DISPLAYCLASS → $\mathcal{P}$(COORD)

retv-display ≙ (λD)

areadC (screenstart,screensize)

The three dimensional area defining function is used to derive the screen from a starting point and a size.

State Component Definition:

DISPLAY4.3A

DISPLAY4.3

DISPLAYCLASS

---

seearea = retv-display (DISPLAYCLASS)

The refined description obviously satisfies the earlier design decisions that the screen should be rectangular and finite, and allows for an easy return to the previous representation.

FILER4A

State Definition:

FILER4A ≙ FILER4
(FILING CABINET4.1 ⤔ FILING CABINET4.1A;
PROJSET4.2 ⤔ PROJSET4.2A;
DISPLAY4.3 ⤔ DISPLAY4.3A)

The new state is simply a combination of the refined state components.

FILER5

Basis:

Prior: FILER4A

Forward: FILING CABINET5.1 / PROJSET5.2 / DISPLAY5.3

Comments:

The current design allows the user to specify any document in the filing cabinet as the document he wishes to view. This advancement of the design will restrict the user to viewing one specific, permanent document in the filing cabinet called `tabletop`. This permanent viewing document can be most easily envisioned as a flat tabletop on which various documents are arranged (projected) for viewing. As before, any combination of the other documents in the filing cabinet can be seen; but now the proper projections of the required documents to the `tabletop` must be defined.

Because of the restriction to view only the tabletop document, we can distinguish between two different types of projections: building projections, which share information between documents, and viewing projections, which define the screen document. Since the screen appearance will be directly derived from this later type of projection, only viewing projections will ever need filter application areas.

FILING CABINET5.1

Basis:

Prior: FILING CABINET4A

Comments:

The filing cabinet will be required to always contain the document which will be displayed (in part) to the user. This document is called the tabletop because it is similar in usage to a tabletop upon which papers have been arranged by the user.

Requirements / Design Decisions:

[[1]] The screen document must be permanent in the FILING CABINET.

Auxiliary Definitions:

a. tabletop ϵ docname

Define tabletop to be a member of the abstract set of allowable document names.

State Component Definition:

FILING CABINET5.1

FILING CABINET4.1A

[[1]] tabletop ϵ dom (docdata)

The FILING CABINET will always contain the document called tabletop.

PROJSET5.2

Basis:

Prior: PROJSET4.2A

Comments:

This level of the PROJSET specification will begin to differentiate between two different types of projections -- the *building* projections and the *viewing* projections. Since the user only sees documents arranged on the `tabletop`, only the projections to the `tabletop` (viewing projections) ever need to have a filter. Therefore, we can specify that projections which do not have a slave on the `tabletop` (building projections) will not have a filter area. We also wish to eliminate the `tabletop` as a projection master because doing so will remove the possibility of unnecessary circularity without a loss of capability.

Requirements / Design Decisions:

[[1]] The `tabletop` is never the master in a projection.

[[2]] Building projections have no requirement for filtering of the slave.

State Component Definition:

PROJSET5.2

PROJSET4.2A

```
      (∀p : ran (projdata))
[[1]]     (dn (masterspot (p)) ≠ tabletop ∧
[[2]]     (dn (slavespot (p)) ≠ tabletop ⇒
                panesize = (0,0)))
```

The state component is restricted as required by the design decisions.

DISPLAY5.3

Basis:

Prior: DISPLAY4.3A

Comments:

We can now require that the user select only a portion of the tabletop document to view.

Requirements / Design Decisions:

[[1]] The screen displays (part of) the screen document (tabletop) only.

State Component Definition:

DISPLAY4

DISPLAY4.3A

[[1]] dn (screenstart) = tabletop

FILER5

State Definition:

$$\begin{aligned}
&\text{FILER5} \triangleq \text{FILER4} \\
&\quad (\text{FILING CABINET4.1A} \rightarrowtriangle \text{FILING CABINET5.1}; \\
&\qquad \text{PROJSET4.2A} \rightarrowtriangle \text{PROJSET5.2}; \\
&\qquad \text{DISPLAY4.3A} \rightarrowtriangle \text{DISPLAY5.3})
\end{aligned}$$

The refined FILER state space is a simple restriction of the previous state space definition.

Of course, tabletop is a derived component since in FILER1 we defined what the *appearance* of every coordinate potentially visible to the user would be. Consequently,

$$\begin{aligned}
&(\forall S : \text{FILER5}) \\
&\text{docdata1 } (S)\ (\text{tabletop}) = \text{fileview}' \\
&\quad (\text{FILE-APPEARANCE}_{\text{FILER5}}\ [\text{FILER5}/S]; \\
&\quad\ \text{PROJECTED-APPEARANCE}_{\text{FILER5}}; \\
&\quad\ \text{APPEARANCE}_{\text{FILER5}})
\end{aligned}$$

FILER6

Basis:

Prior: FILER5

Forward: FILING CABINET6.1 / PROJSET6.2 / DISPLAY6.3

Comments:

This specification level defines a number of size and extent limitations. Many of these limitations will be only abstractly defined so that actual values can be given to them after additional functional decisions and non-functional requirements have been defined.

FILING CABINET6.1

Basis:

Prior: FILING CABINET5.1

Comments:

The limitations to be placed on the filing cabinet includes a maximum allowable number of documents in the filing cabinet. Additionally, the filing cabinet can be simplified by restricting the background colors available.

Requirements / Design Decisions:

[[1]] The background *paper color* for all documents and the filing cabinet background color will be the same.

[[2]] The `tabletop` must have a maximum size.

[[3]] All non-viewing documents must have a maximum size.

[[4]] The filing cabinet has a maximum number of documents that it can hold.

Auxiliary Definitions:

We define abstract limits for the filing cabinet here.

a. `backcolor` ϵ `speck`

b. `tablelength` ϵ $\mathbb{N}$
   `tablewidth` ϵ $\mathbb{N}$

c. `maxdoclength` ϵ $\mathbb{N}$
   `maxdocwidth` ϵ $\mathbb{N}$

d. `maxdocno` ϵ $\mathbb{N}$

State Component Definition:

FILING CABINET6.1

FILING CABINET5.1

---

[1] (∀p : ran (colors)) (p = backcolor)

[1] filecolor = backcolor

[2] rows (docdata (tabletop)) = tablelength
cols (docdata (tabletop)) = tablewidth

[3] (∀d : dom (docdata) - {tabletop})
(rows (docdata (x)) ≤ maxdoclength ∧
cols (docdata (x)) ≤ maxdocwidth)

[4] card (dom (docdata)) ≤ maxdocno

The desired limitations are placed on the previous state component definition.

PROJSET6.2

Basis:

Prior: PROJSET5.2

Comments:

The viewing projections have been restricted here to filters of reverse video (dark on light) only. This decision is based on the normal capability of the textual display screen likely to be used with this system. Also, The maximum number of projections allowed in the system has been limited.

Requirements / Design Decisions:

[[1]] The only filter available will be *reverse video*.

[[2]] The total number of projections allowed is limited.

Auxiliary Definitions:

a. revvideo $\in$ filter

b. maxproj $\in \mathbb{N}$

State Component Definition:

PROJSET6.2

PROJSET5.2

[[1]] $(\forall p : ran\ (projdata))$
$(revvideo = retv\text{-}filter\ (f\ (p)))$

[[2]] $length\ (projdata) \leq maxproj$

DISPLAY6.3

Basis:

Prior: DISPLAY5.3

Comments:

This advancement commits the design to a constant sized screen. Since almost all textual screens are of fixed length and width, this is not a significant restriction of capability.

Requirements / Design Decisions:

[[1]] The screen on which the user sees the tabletop will have a constant size.

Auxiliary Definitions:

a. screenlength ϵ $\mathbb{N}$
screenwidth ϵ $\mathbb{N}$

State Component Definition:

DISPLAY6.3

DISPLAY5.3

[[1]] screensize = (screenwidth, screenlength)

Comments (con't):

Here we will require that projections link documents which exist within the filing cabinet. This will impose a policy of projection deletion when a master or slave document is deleted. We also require that the area displayed to the user is totally in the limited extent of the tabletop.

Requirements / Design Decisions:

[[1]] The slave and master documents of all projections must exist in the filing cabinet.

[[2]] The master of all projections must be entirely on the paper of the master document.

[[3]] The slave of viewing projections will stay on the tabletop.

[[4]] The area of the screen must be entirely on the tabletop.

State Definition:

FILER6

FILER5(FILING CABINET5.1 ⤔ FILING CABINET6.1;
PROJSET5.2 ⤔ PROJSET6.2;
DISPLAY5.3 ⤔ DISPLAY6.3)

[[1]] (∀p : ran (projdata))
(dn (masterspot (p)) ∈ dom (docdata) ∧
dn (slavespot (p)) ∈ dom (docdata) ∧

[[2]] strip (areaC (masterspot (p), projsize)) ⊆
(cols (docdata (dn (masterspot (p)))),
rows (docdata (dn (masterspot (p))))))

[[3]] (∀p : ran (projdata) |
dn (slavespot (p)) = tabletop)
(strip (areaC (slavespot (p), projsize)) ⊆
area ((1,1), (tablewidth,tablelength)))

[[4]] aread (coord (screenstart)), screensize) ⊆
area ((1,1), (tablewidth,tablelength))

FILER7

Basis:

Prior: FILER6A

Forward: FILING CABINET7.1 / PROJSET7.2

Comments:

With this specification we begin to include the lower level details of the desired system by introducing another specialized document called homedoc. This permanent document contains the significant information the user wishes to supply about each user defined document in the filing cabinet (ie. the namelines of the documents.) The homedoc will also hold information about the length of the documents. The restrictions imposed here are all required to define the necessary home document.

FILING CABINET7.1

Basis:

Prior: FILING CABINET6.1

Comments:

This advancement places the homedoc permanently in the filing cabinet and defines its size.

Requirements / Design Decisions:

[[1]] The *home* document is always in the filing cabinet.

[[2]] The width of the home document is the same as the maximum allowable document width.

[[3]] The length of the home document is determined by the number of documents in the filing cabinet excluding the tabletop document.

Auxiliary Definitions:

a. homedoc ∈ docname

State Component Definition:

FILING CABINET7.1

FILING CABINET6.1

[[1]] homedoc ∈ dom (docdata1)

[[2]] cols (docdata (homedoc)) = maxdocwidth

[[3]] rows (docdata (homedoc)) = 2 *
(card (dom (docdata)) - 1)

PROJSET7.2

Basis:

Prior: PROJSET6.2 / FILING CABINET7.1 (Auxiliary Definitions)

Comments:

Now that we have the document namelines gathered together on the homedoc we do not want the appearance of the homedoc to be spoiled by projections to it. Therefore, we eliminate the possibility of the homeline document ever being the slave of a projection. We also specify that every *normal* window on the tabletop must have its appropriate `nameline` at the top to identify the document being displayed. In order to protect the user from *misplacing* windows by making them too small to see, we require that part of the nameline must always show, even if the window it tops is too small to be visible.

(Since viewing *projection* slaves appear like *windows* onto a document, we shall use that term whenever discussing tabletop areas which contain a document projection.)

Requirements / Design Decisions:

[[1]] The home document is never the slave in a projection.

[[2]] For each viewing projection which does not have the home document as its master, there is immediately following it in the projection list a *nameline* projection from the home document to the top of the viewing projection window.

[[3]] Namelines always appear in reverse video at the top of a window.

[[4]] A projection must always be at least one column wide so that its nameline will never disappear from the screen.

State Component Definition:

PROJSET7.2

PROJSET6.2

[[1]] (∀p : ran (projdata)) (dn (slavespot (p)) ≠ homedoc

[[4]] ∧ dn (slavespot (p)) = tabletop ⇒

projsize (p) > (1,0))

[[2 & 3]] (∀p : ran (projdata) |

dn (masterspot (p)) ≠ homedoc)

∧ dn (slavespot (p)) = tabletop)

(q = p + 1 ∧

dn (masterspot (q)) = homedoc ∧

dn (slavespot (q)) = tabletop ∧

coord (slavespot (q)) =

coord (slavespot (p)) - (1,0) ∧

retv-filter (f (q)) = revvideo ∧

projsize (q) = (x (projsize (p)), 1) ∧

startpane (q) =

coord (slavespot (p)) - (1,0) ∧

panesize (q) = (x (projsize (p)), 1))

Notice that the masterspot for the nameline projection has not been defined here. It can be more easily specified in the FILER7 definition which follows.

Comments (con'd):

In this specification, we had to add a new component to the definition of FILER. This new component defines the relationship between the existent filing cabinet documents and the lines of the home document which contain the namelines. This simple function will help build the nameline projection required with each of the viewing projections.

Requirements / Design Decisions:

[[1]] For each non-tabletop document a unique entry of nameline and document length is made on homedoc.

[[2]] The home document nameline is the first entry in the home document.

Auxiliary Definitions:

a. marker ϵ speck

This will abstractly define the speck to be used as a document length indicator on the homedoc. Each occurrence of the marker represents 10 lines in the document except for the last occurrence which could represent any number of lines up to (and including) ten.

b.

odds $\triangleq$ $\{2n + 1 \mid n : \mathbb{N}_1\}$

The odds set will define the lines of the homedoc which contain namelines. The remaining (even) lines of the homedoc contain the document length information appropriate to the document described by the previous nameline.

State Component Definition:

```
FILER7 ──────────────────────────────────────────────
      FILER6
          (FILING CABINET6.1 ⋈ FILING CABINET7.1
                  PROJSET6.2 ⋈ PROJSET7.2)
      point        :        docname >→ odds
      ──────
[[1]] dom (point) = dom (docdata) - {tabletop}
          ∧ ran (point) =
                {a : odds | a < 2*(card(docdata)-1)}

[[1]] (∀d : dom (point))
            (∃x : SEQ [ {marker,backcolor} ] |
                card (x ↓ {marker}) =
                  (rows (docdata (d)) + 9) DIV 10 ∧
                x = homedoc (point (d) + 1))

[[2]] point (homedoc) = 1

      (∀p : ran (projdata) |
          dn (masterspot (p)) ≠ homedoc ∧
          dn (slavespot (p)) = tabletop)
              (q - p + 1 ∧
               coord (masterspot (q)) =
                   (point (dn (masterspot (p))), 1))
```

This new filer definition creates the nameline document and the relation which connects the documents to their respective namelines. Notice that the nameline projections partially specified in PROJSET7.2 is completed here with the definition of where to find the required nameline.

FILER7A

Basis:

Prior: FILER7

Forward: FILING CABINET7.1A / PROJSET7.2A / DISPLAY7.3A

Comments:

This refinement will simplify the state space by removing the tabletop document from the list of filing cabinet documents. Since it is a highly specialized document and is in general not subject to the same restrictions and operations as the other documents, its disassociation from the rest of the documents serves to eliminate some of the previously required state predicates and operation preconditions.

Similarly, we can create separate projection lists for the viewing and building projections; thereby, facilitating the definition of required operations on the *different projection* types. At the same time, we will eliminate from the projection list the explicit (but derived) `nameline` viewing projections which are used to top the normal viewing projections.

FILING CABINET7.1A

Basis:

Prior: FILING CABINET7.1

Comments:

We can now easily remove the tabletop from the rest of the documents in the filing cabinet.

State Component Definition:

FILING CABINET7.1A

FILING CABINET7.1

tabletop : DOCUMENTCLASS

docdata1 : docname ⇸ DOCUMENTCLASS

---

tabletop ∉ dom (docdata1)

docdata = docdata1 ⊕ {tabletop ↦ tabletop}

The relationship between the old state component and the new state component is straightforward.

PROJSET7.2A

Basis:

Prior: PROJSET7.2

Comments:

Here we have created different schemata for the two different types of projections, allowing the elimination of the unneeded derived components of the projections. The separation of the projections into two lists does not alter the ordered property of the projection list (ie. relative projection strength) because the two types of projections never have slaves on the same document.

Auxiliary Definitions:

a.

PROJCLASSB

slavespot	:	COORD
masterspot	:	COORD
projsize	:	coordinate

---

projsize > (0,0)

The new building projection definition makes use of the fact that no filter or area of filter application (which we will call *stained glass* from now on) is required on this type of projection.

b.

PROJCLASSV

slavespot	:	coordinate
masterspot	:	COORD
projsize	:	coordinate
startpane	:	coordinate
panesize	:	coordinate

---

projsize ⩾ (1,0)
panesize ⩾ (0,0)
aread (startpane,panesize) ⊆
aread (slavespot,projsize)

Viewing projections have a predetermined slave document (tabletop) and filter (revvideo) so these derived components can be removed from the *projection definition*. Of course, the stained glass area must still remain on the window and the window must always have some width.

c.

maxvproj ∈ $\mathbb{N}$
maxbproj ∈ $\mathbb{N}$

We can replace the general restriction on the number of projections by individual maximums on the two different projection types.

d.

```
retv-vprojection : PROJCLASSV → (coordinate ⇸ COORD)
retv-vprojection ≙ (λP)
                    (λc | c ∈
                       aread (slavespot(P),projsize(P)))
                   (dn(masterspot(P)),
                   (coord(masterspot(P))+c-slavespot(P)))
```

This function recreates a viewing projection from a PROJCLASSV schema.

```
e.
retv-bprojection : PROJCLASSB → projection
retv-bprojection ≡ (λP)
                    (λC | dn(C) = dn(slavespot(P)) ∧
                       coord(C) ∈
aread(coord(slavespot(p)),
                          projsize(p)))
                    (dn(masterspot(P)),
                       (coord(masterspot(P))+coord(C)-
                          coord(slavespot(P))))
```

Now we can retrieve a building projection from its new schema representation.

State Component Definition:

```
PROJSET7.2A
      PROJSET7.2
      projbuild    :        SEQ [PROJCLASSB]
      projview     :        SEQ [PROJCLASSV]
-----------

      length (projbuild) ≤ maxbproj
      length (projview) ≤ maxvproj

      (∀i,j : dom(projdata) |
          dn(slavespot(projdata(i))) ≠ tabletop ∧
          dn(slavespot(projdata(j))) ≠ tabletop ∧
          i < j)
            (∃p,q : dom(projbuild) |
                p < q ∧
                retv-projection(projdata(i)) =
                   retv-bprojection(projbuild(p)) ∧
                retv-projection(projdata(j)) =
                   retv-bprojection(projbuild(q)))

      (∀i,j : dom(projdata) |
          dn(slavespot(projdata(i))) =  tabletop ∧
          dn(slavespot(projdata(j))) =  tabletop ∧
          i < j)
            (∃p,q : dom(projview) |
                p < q ∧
                retv-projection(projdata(i))(tabletop,c) =
                   retv-bprojection(projview(p))(c) ∧
                retv-projection(projdata(j))(tabletop,c) =
                   retv-bprojection(projview(q))(c))

      length(projdata/(PROJCLASS[dn(masterspot/homedoc])))
      =
      length(projbuild) +
      length(projview/(PROJCLASSV[dn(masterspot/homedoc]))
```

Since a projection from the building projection list cannot possibly cause a slave overlap with a projection defined in the viewing projection list, the relative order of the projections between the two lists is unimportant. That is, any recombination of the two lists which preserves their individual orderings will produce a list equivqlent to the original projection list.

Operations:

The following operations adhere to the previously defined policy of using composition to move or size objects where possible. With projections defined by starting points and sizes, the addition of an offset becomes the functional equivalent of composition. All but the final operation involves projections of the viewing type.

a.

```
movewindow : PROJCLASSV × coordinate → PROJCLASSV
movewindow ≙ (λP,o)
             PROJCLASSV [ slavespot/slavespot(P) + o;
                          masterspot/masterspot(P);
                          projsize/projsize(P);
                          startpane/startpane(P) + o;
                          panesize/panesize(P) ]
```

Moves the window and its associated stained glass according to the given offset...

b.

```
movesg : PROJCLASSV × coordinate ⇸ PROJCLASSV
movesg ≙ (λP,o | aread(startpane(P)+o,panesize(P))
                      ⊆ aread(slavesspot(P),projsize(P)))
         PROJCLASSV [ slavespot/slavespot(P);
                      masterspot/masterspot(P);
                      projsize/projsize(P);
                      startpane/startpane(P) + o;
                      panesize/panesize(P) ]
```

Moves the stained glass by the offset amount.... The stained glass must remain within the window.

c.

```
sizewindow : PROJCLASSV × ℕNT × ℕNT × ℕNT × ℕNT
                                  ⇸ PROJCLASSV
sizewindow ≙ (λP,u,d,l,r |
                        projsize(P)+(r,d)+(l,u) ≥ (1,0))
             PROJCLASSV [ slavespot/slavespot(P) - (l,u);
                          masterspot/masterspot(P) -
                                                (l,u);
                          projsize/projsize(P) +
                                 (r,d) + (l,u);
                          startpane/startpane(P);
                          panesize/panesize(P) ]
```

Adjusts the size of the window by adjusting the location of each of the window sides.... The projection nameline must still be visible after the size change.

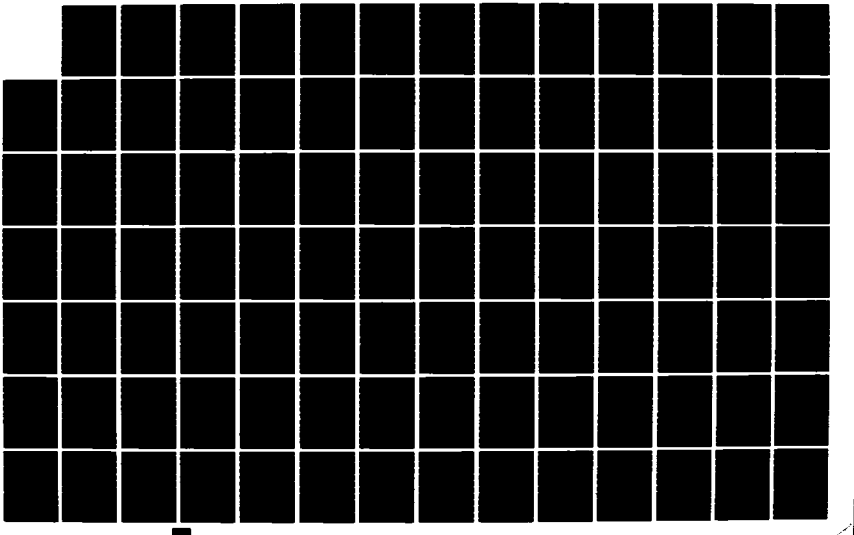
d.

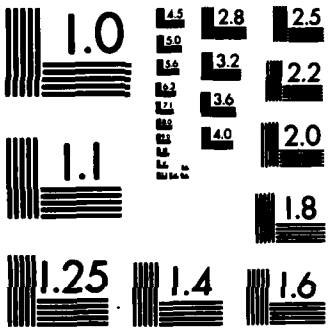
```
sizesg : PROJCLASSV × ℕNT × ℕNT × ℕNT × ℕNT
                                  ⇸ PROJCLASSV
sizesg ≙ (λP,u,d,l,r | aread(startpane(P)-(l,u),
                         panesize(P)+(r,d)+(l,u)) ⊆
                       aread(slavespot(P),projsize(P)) ∧
                       panesize(P)+(r,d)+(l,u) ≥ (0,0))
           PROJCLASSV [ slavespot/slavespot(P);
                        masterspot/masterspot(P);
                        projsize/projsize(P);
                        startpane/startpane(P) - (l,u);
                        panesize/panesize(P) +
                                  (r,d) + (l,u) ]
```

To change the size of the stained glass area.... The stained glass will remain on the window and not have a negative size.

AD-A132 569 FORMAL TECHNIQUES IN THE MANAGEMENT OF SOFTWARE DESIGN 3/4
(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH
W E RICHARDSON 17 JUN 83 AFIT/CI/NR-83-28D

UNCLASSIFIED F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

e.

```
removesg : PROJCLASSV → PROJCLASSV
removesg ≙ (λP)
           PROJCLASSV [ slavespot/slavespot(P);
                        masterspot/masterspot(P);
                        projsize/projsize(P);
                        startpane/startpane(P);
                        panesize/(0,0) ]
```

This will remove the stained glass.

f.

```
createsg : PROJCLASSV × coordinate × coordinate ⇸
                                          PROJCLASSV
createsg ≙ (λP,startc,stopc | startpane(P) = (0,0)
                    ∧ startc ≤ stopc)
           PROJCLASSV [ slavespot/slavespot(P);
                        masterspot/masterspot(P);
                        projsize/projsize(P);
                        startpane/startc;
                        panesize/stopc-startc+(1,1) ]
```

To create a stained glass area, there must not be such an area associated with the projection already and the diagonal of the desired stained glass area must be properly defined.

g.

```
scrolldoc : PROJCLASSV × coordinate → PROJCLASSV
scrolldoc ≙ (λP,o)
            PROJCLASSV [ slavespot/slavespot(P);
                         masterspot/
                             (dn(masterspot(P)),
                              coord(masterspot(P))+o);
                         projsize/projsize(P);
                         startpane/startpane(P);
                         panesize/panesize(P) ]
```

The master of a projection can be moved with an offset.

h.

$$\text{swap-projection} : \text{SEQ}\ [\ \text{PROJCLASSV}\ ] \times \mathbb{N}_1 \times \mathbb{N}_1 \rightarrow\!\!\!\!\!\!\rightarrow \text{SEQ}\ [\ \text{PROJCLASSV}\ ]$$

$$\text{swap-projection} \triangleq (\lambda P,m,n \mid m \in \text{dom}(P) \wedge n \in \text{dom}(P))\ P \oplus \{m \mapsto P(n)\} \oplus \{n \mapsto P(m)\}$$

The strengths of two existing projections in the viewing projection list are swapped by this function.

The following function deals with building projections.

i.

$$\text{createproj} : \text{COORD} \times \text{coordinate} \times \text{COORD} \rightarrow\!\!\!\!\!\!\rightarrow \text{PROJCLASSB}$$

$$\text{createproj} \triangleq (\lambda \text{cloc},\text{size},\text{ploc} \mid \text{size} > (0,0))\ \text{PROJCLASSB}\ [\ \text{slavespot/ploc};\ \text{masterspot/cloc};\ \text{projsize/size}\ ]$$

This function creates a building projection from its components.

DISPLAY7.3A

Basis:

Prior: DISPLAY7.3

Comments:

For the sake of simplicity, we rename the only non-derived portion of the DISPLAY state component.

State Component Definition:

DISPLAY7.3A

DISPLAY7.3

sstart : coordinate

---

sstart = coord (screenstart)

FILER7A

State Definition:

FILER7A

FILER7 (FILING CABINET7.1 ⊳⊲ FILING CABINET7.1A;
PROJSET7.2 ⊳⊲ PROJSET7.2A;
DISPLAY6.3 ⊳⊲ DISPLAY7.3A)

The state space is the previous state space as refined by this level of specification.

UMACHINE

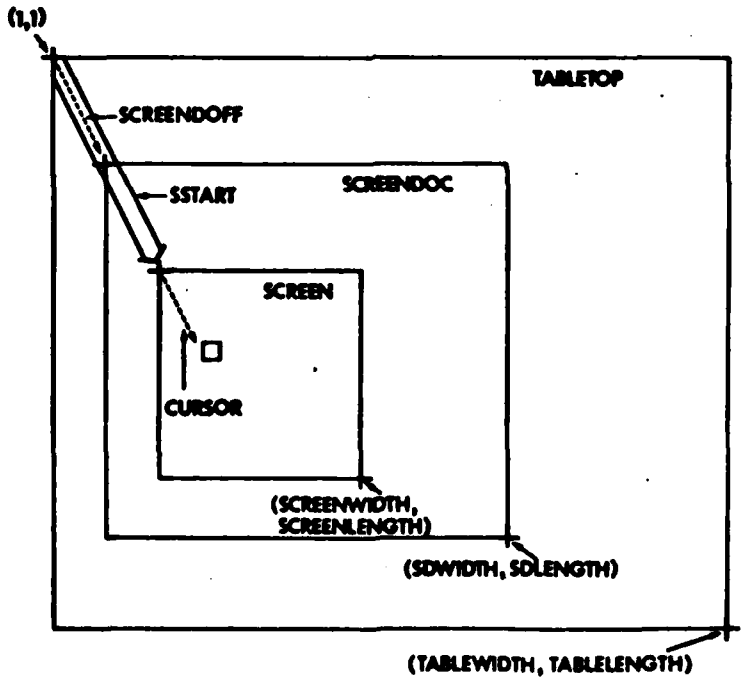
Basis:

Prior: FILER7A / EDDOC1A

Comments:

This specification recombines the decomposed system parts -- the filer and the editor. Also at this level, many of the user required commands can be initially formulated. In order to facilitate the development of these commands, a number of new components have been added to the state space. Many of these new components are derived from the current state parts but provide information upon which decisions will be made within the commands.

Finally, this specification defines a subset of the tabletop document in order to save machine resources. Typically, this partial tabletop is defined by a starting point (called the screen document offset, `screendoff`) and a size parameter. This partial tabletop represents the first decision made about how the screen appearance may be created and stored in our system. It is hopelessly unrealistic to expect that the tabletop appearance would be recomputed in toto after each state changing operation. However, it may be economical to compute the appearance of an area larger than just the area seen by the user. The tabletop section selected for appearance computation is called the `screendoc`. The following figure shows the relationship among the various tabletop document components.



The screen document must remain on the tabletop document, the screen must remain on the screendoc and the cursor must remain on the screen.

Requirements / Design Decisions:

[1] A partial tabletop appearance is all that could be economically computed when the user's display changes as a result of operations on the state space. Defining this partial tabletop larger than the screen area may save resources so we leave the option open.

[2] Decisions about required user commands noted in the provisional User's Manual are reflected here.

[3] No editing operations (except speck replacement) will be allowed on document lines which are partially determined by building projections. This eliminates some potentially undesirable effects caused by the interaction of projected and non-projected information (eg. words disappearing *under* a projection during an insert.)

Auxiliary Definitions:

a.

```
blankline ≙ SEQ [ {backcolor} ]
length (blankline) = maxdocwidth
blankmsg ≙ SEQ [ {backcolor} ]
length (blankmsg) = screenwidth
```

b.

```
{newdoc,blankdoc} ⊆ docname
```

A new document which has not yet been filed permanently and a blank document will be required in the filing cabinet, so the appropriate names are added to the abstract set of document names.

c.

```
blankproj ≙ PROJCLASSV[ slavespot/(1,2);
                        masterspot/(blankdoc,(1,1));
                        projsize/
                           (maxdocwidth,maxdoclength);
                        startpane/(1,2),
                        panesize/(0,0) ]
```

This projects a very large blank document to the upper left corner of the tabletop. This document will serve as the backdrop for the creation of a new document.

```
d.
nullproj ≙ PROJCLASSV[ slavespot/(1,2);
                       masterspot/(newdoc,(1,1));
                       projsize/(1,0);
                       startpane/(1,2);
                       panesize/(0,0) ]
```

This will initialize the viewing projection for the creation of a new document.

```
e.
blankdoc ≙ DOCUMENTCLASS[ rows/maxdoclength;
                          cols/maxdocwidth |
                          (∀r : 1..maxdoclength)
                          (∀c : 1..maxdocwidth)
                          (table (r)(c) = backcolor) ]
```

This is the big blank document which will hide the tabletop from the user as he creates a new document.

```
f.
homeproj ≙ PROJCLASSV[ slavespot/(1,1);
                       masterspot/(homedoc,(1,1));
                       projsize/
                  (maxdocwidth,length(docdatal(homedoc)));
                       startpane/(1,1);
                       panesize/(maxdocwidth,1) ]
```

To display the home document, this projection will be added to the end of the viewing projection list.

```
g.
emptydoc ≙ DOCUMENTCLASS [ table/<> ]
```

h.

CAMERA

loc	:	COORD
size	:	coordinate

---

size > (0,0)

This schema represents the master area of a building projection.

i.

none $\notin$ docname

edocname = docname $\cup$ {none}

j.

sdwidth $\in \mathbb{N}$

sdlength $\in \mathbb{N}$

These abstract values define the size of the screen document.

k.

oddit : $\mathbb{N}_1 \rightarrow \mathbb{N}_1$

oddit = (λa)

a + a MOD 2 - 1

Oddit turns even and odd line numbers into odd line numbers.

l.

The following functions are used to insure that building projections will not be disturbed by the editing commands.

check : docname $\times$ $\mathbb{N}$ $\times$ $\mathbb{N}_1$ $\times$

SEQ [PROJCLASSB] $\rightarrow$ $\mathbb{P}$ ($\mathbb{N}_1$)

check = (λd,width,line,B)

{c : dom (B) | ($\exists$i : 1..width |

(i,line) $\in$

aread (coord(slavespot(B(c))),projsize(b(c)))

$\wedge$ d = dn(slavespot(B(c))))}

The function check defines the set of all building projections which have a slave area that includes some portion of the given line.

$$\begin{aligned}
&\text{lineok} : \text{docname} \times \mathbb{N} \times \mathbb{N}_1 \times \\
&\qquad \text{SEQ [PROJCLASSB]} \rightarrow \text{boolean} \\
&\text{lineok} \triangleq (\lambda d, \text{width}, \text{line}, B) \\
&\qquad \text{check}\ (d, \text{width}, \text{line}, B) = \Phi
\end{aligned}$$

Lineok indicates that a document line is free of building projections to it.

$$\begin{aligned}
&\text{linesok} : \text{docname} \times \mathbb{N} \times \mathbb{N}_1 \times \\
&\qquad \text{SEQ [PROJCLASSB]} \rightarrow \text{boolean} \\
&\text{linesok} \triangleq (\lambda d, \text{width}, \text{line}, B) \\
&\qquad \text{check}\ (d, \text{width}, \text{line}, B) \cap \\
&\qquad \text{check}\ (d, \text{width}, \text{line}+1, B) = \Phi
\end{aligned}$$

This function determines if a line and its successor (if any) share a building projection slave.

$$\begin{aligned}
&\text{areaok} : \text{docname} \times \mathbb{N} \times \mathbb{N}_1 \times \mathbb{N}_1 \times \\
&\qquad \text{SEQ [PROJCLASSB]} \rightarrow \text{boolean} \\
&\text{areaok} \triangleq (\lambda d, \text{width}, \text{startl}, \text{stopl}, B) \\
&\qquad \bigcup_{(n=\text{startl}..\text{stopl})} \text{check}\ (d, \text{width}, n, B) = \Phi
\end{aligned}$$

Finally, an entire area (sequence of lines) is tested by this function to see if it overlaps the slave area of any building projection.

m.

$$\begin{aligned}
&\text{innameline} : \text{SEQ [PROJCLASSV]} \times \mathbb{N} \times \text{coordinate} \times \\
&\qquad \text{coordinate} \rightarrow \text{boolean} \\
&\text{innameline} \triangleq (\lambda P, \text{vpno}, \text{cursor}, \text{sstart}) \\
&\qquad (\text{vpno} \neq 0) \wedge \\
&\qquad (\text{vpno} \neq 0) \Rightarrow \text{cursor}+\text{sstart}-(1,1) \in \\
&\qquad\quad \text{aread}(\text{slavespot}(P(\text{vpno}))-(0,1), \\
&\qquad\qquad (x(\text{projsize}(P(\text{vpno}))),1))
\end{aligned}$$

It will be important at times to know if the cursor is in the nameline of a projection. This function will determine that information.

n.

The deletion of a document may require that projections referencing that document be deleted.

```
deletepv : SEQ [ PROJCLASSV ] → docname
                              → SEQ [ PROJCLASSV ]
deletepv ≙ (λS)
           (λd)
           (ns | ran (ns) = {t : ran (S) |
                          dn(masterspot(t)) ≠d}
             ∧ (∀i,j : dom (ns)) (∃p,q : dom (S) |
                          i > j  ⇒
                               p > q ∧
                               ns(i) = S(p) ∧
                               ns(j) = S(q))
             ∧ length (ns) =
                    length (S ↓ (PROJCLASSV |
                               dn(masterspot) ≠ d )))
```

With this function we can delete all viewing projections which have the given document as the master document.

```
deletepb : SEQ [ PROJCLASSB ] → docname
                              → SEQ [ PROJCLASSB ]
deletepb ≙ (λS)
           (λd)
           (ns | ran (ns) = {t : ran (S) |
                          dn(masterspot(t)) ≠d ∧
                          dn(slavespot(t)) ≠d}
             ∧ (∀i,j : dom (ns)) (∃p,q : dom (S) |
                          i > j ⇒
                               p > q ∧
                               ns(i) = S(p) ∧
                               ns(j) = S(q))
             ∧ length  ns) =
                    lengt  (S ↓ (PROJCLASSB |
                               dn(masterspot) ≠ d ∧
                               dn(slavespot) ≠ d)))
```

We have created a function which will delete building projections which have the given document as either the master or slave.

o.

```
vprojreduce : SEQ [ PROJCLASSV ] × docname × ℕ
              → SEQ [ PROJCLASSV ]
vprojreduce ≙ (λS,d,n)
               (ns | (∀x : 1..length(S))
                ((dn(masterspot(S(x))) ≠ d ∨
                   dn(masterspot(S(x))) = d ∧
                      y(coord(masterspot(S(x))) +
                      y(projsize(S(x)))-(1,1) ≤ n ⇒
                          ns(x) = S(x))
                 ∧ dn(masterspot(S(x))) = d ∧
                      y(coord(masterspot(S(x))) +
                      y(projsize(S(x)))-(1,1) >  n ⇒
                          ns(x) =
             PROJCLASSV [ slavespot/slavespot(S(x));
                          masterspot/masterspot(S(x));
                          projsize/(x(projsize(S(x))),
                         n+1-y(coord(masterspot(S(x)))));
                          startpane/startpane(S(x));
                          panesize/
                         (x(panesize(S(x))),
                         min({y(panesize(S(x))),
                         n+1-y(coord(masterspot(S(x))))}) ]
```

In order to keep the master of a projection on its master document it may be necessary to decrease projection sizes if documents participating in those projections are decreased in size. Of course the pane sizes may have to be reduced too. Those projections which do not involve the reduced document or which still fit on the document even after the document is reduced will not be affected by this function.

The auxiliary definition for bprojreduce is similar to the above except that it tests and produces a PROJCLASSB.

p.

tabletopdisplay is ABSTRACT

This represents the policy for moving the screen and for automatic disclosure required to insure that user commands do not force the cursor out of the current window or off the screen. This policy may include the swapping of projection strengths to uncover parts of a window or movement of the screen on the tabletop (screendoc).

```
q.
barupdate : ℕ₁ × docname × DOCUMENTCLASS × ℕ
                                  ⇸ DOCUMENTCLASS
barupdate ≙ (λline,d,H,long | line ≤ length(H))
            barlength < long ⇒
               iter(enter,((long-barlength+9) DIV 10))
                                        (H(line))
            barlength ≥ long ⇒ H

where barlength ≙ 10 * (endline(line,H)-1)

      enter : SEQ [ speck ] → SEQ [ speck ]
      enter (S) = ⟨marker⟩ * head (S)
```

The document lengths recorded in the home document must be updated each time a document's length is changed. This function updates these length bars as necessary.

```
r.
insertel : DOCUMENTCLASS × ℕ₁ × coordinate
                                  → DOCUMENTCLASS
insertel ≙ (λd,l,c | y(c) ∈ dom(d) ∧
                                x(c) ∈ dom(d(y(c))))
           paste (d,y(c),{l ↦ (blankline ⊕
              (shift (l-1); (iter(tail,x(c)-1)(d(y(c))))
                                ↑ (1..cols(d))}))
           ⊕ {y(c) ↦ blankline ⊕
              iter(head,length(d(y(c)))-x(c)+1)(d(y(c)))
                                ↑ (1..cols(d))}
```

This function divides a document line into two parts to create a new line in the document. The remainder of the two new lines is filled with the background color.

s.

```
deleteel : DOCUMENTCLASS × ℕ₁ × coordinate
                                    → DOCUMENTCLASS
deleteel ≙ (λd,l,c | y(c) ∈ dom(d) ∧
                     x(c) ∈ dom(d(y(c))) ∧
                     y(c) < length(d))
          cut (d ⊕ {y(c) ↦
              iter(head,length(d(y(c)))-x(c)+1)(d(y(c)))
            * iter(tail,l-1)(d(y(c)+1)) ↑ (1..cols(d))},
              y(c),1)
```

With this function we can compress two document lines into one line of the proper length.

State Definition:

```
UMACHINE
      FILER7A
      EDDOC1A
      proj          :       COORD
      cam           :       CAMERA
      cursor        :       coordinate
      screendoff    :       coordinate
      insertmode    :       boolean
      arrowcommand  :       boolean
      start         :       coordinate
      stop          :       coordinate
      msgline       :       SEQ [ speck ]

%%% USEFUL DERIVED COMPONENTS
      whichdoc      :       edocname
      show          :       {newdoc,homedoc,tabletop}
      nodoc         :       ℕ₁
      novproj       :       ℕ
      nobproj       :       ℕ
      vpno          :       ℕ
      bpno          :       ℕ
      svpno         :       ℕ
      screendoc     :       DOCUMENTCLASS
```

```
length (msgline) ≤ screenwidth - 1

%%% fixes blankdoc permanently in the filing cabinet.
blankdoc ∈ dom (docdatal)

%%% although visible, the cursor can't be in a blankdoc
window.
dn (masterspot (z)) ≠ blankdoc

%%% the doc. to be edited is the current document.
whichdoc ≠ none ⇒
      docdatal (whichdoc) = doc

%%% the blankdoc cannot be cluttered by projections.
(∀b : ran (projbuild))
      (dn (slavespot (b)) ≠ blankdoc)

size (cam) ≥ (0,0)

%%% the cursor stays on the screen.
cursor ∈ area ((1,1), (screenwidth,screenlength))

%%% the screendoc stays on the tabletop
screendoff ∈ area ((1,1),
      (tablewidth-sdwidth+1,tablelength-sdlength+1))

%%% the screen stays on the screendoc.
sstart ∈ aread(screendoff,
      (sdwidth-screenwidth+1,sdlength-screenlength+1))

nodoc = card (docdatal)
novproj = length (projview)
nobproj = length (projbuild)
```

```
%%% the projection number of the window in which the
cursor is -- including the nameline of the window.
vpno ≠ 0 ⟺
      NOT (∃n : vpno+1..novproj | (cursor+sstart-(1,1)) ∈
          aread(slavespot(zz)-(0,1),projsize(zz)+(0,1)))
      ∧ (cursor+sstart-(1,1)) ∈
          aread(slavespot(z)-(0,1),projsize(z)+(0,1))

%%% the building projection which is the last in the
projection network for the cursor location.
bpno ≠ 0 ⟺
      vpno ≠ 0
      ∧ a = {n : 1..nobproj | retv-vprojection(z)
                        (cursor+sstart-(1,1))
                  ∈ dom(retv-bprojection(projbuild(n)))}
      ∧ a ≠ 0
      ∧ bpno = max(a)

%%% the second strongest viewing projection at the current
cursor location.
svpno ≠ 0 ⟺
      vpno ≠ 0
      ∧ NOT(∃n : svpno+1..vpno-1 | (cursor+sstart-(1,1)) ∈
          aread(slavespot(zz)-(0,1),projsize(zz)+(0,1)))
      ∧ (cursor+sstart-(1,1)) ∈
          aread(slavespot(projview(svpno))-(0,1),
                  projsize(projview(svpno))+(0,1))

%%% this picks the current document for the cursor
location if that spot is not in the slave of a building
proj, or in a nameline.
whichdoc ≠ none ⟺
      vpno ≠ 0 ∧
      NOT innameline(projview,vpno,cursor,sstart) ∧
      areaok(whichdoc,cols(docdatal(whichdoc)),
            y(coord(masterspot(z))),
            y(coord(masterspot(z)))+y(projsize(z))-1) ∧
      retv-vprojection(z)
                  (cursor+sstart-(1,1))
                        = (whichdoc,position)
```

```
%%% a quick shorthand of what document the user is seeing.
show = homedoc ⟺
      projview(novproj) = homeproj ∧ whichdoc = homedoc
show = newdoc  ⟺
      newdoc ∈ dom(docdata1)
show = tabletop ⟺
      whichdoc ∉ {homedoc,newdoc}

rows (screendoc) = sdlength
cols (screendoc) = sdwidth

where
      z  ≙ projview (vpno)
      zz ≙ projview (n)
```

Operations:

The following operations will use this delta operation in the initial definition of user commands.

ΔUMACHINE

```
      UMACHINE
      UMACHINE'
      status'      :       STATUS
```

```
      status'  = SUCCESS
```

*** NOTICE: From this specification forward we will institute a shorthand convention of <u>not</u> explicitly showing state components which are not changed in an operation. This convention can safely be applied because there will not, at this or following levels of specfication, be any cases where the result of an operation is unknown for some of the state components (which was the previous meaning given to leaving state components unspecified.) This new convention will greatly simplify the presentation of the operations and highlight their effect on the state.

These user commands correspond with the appropriately named user commands described in the provisional user's manual. Therefore, little or no additional explication about WHAT they do is given here. An explanation of the preconditions or how they insure a transition to a legal state is afforded where required. This explanation will also present a shorthand indication of error conditions required to make them total functions. For example, in the operation DCREATE (operation c.) is the precondition:

%%% there must be room for the new document.(ERROR13)

nodoc < maxdocno

which indicates an error condition of the form:

NODOCROOM?

ΦUMACHINE

status' : STATUS

---

nodoc = maxdocno

status' = ERROR13

The error message associated with ERROR13 is listed at the end of the specification. Of course, the total operation would be written in the form:

DCREATE ≙ DCREATE ∨ NODOCROOM? ∨ ...

A few of the preconditions do not have an error status associated with them. This signifies that the precondition will always be trivially met in the system and therefore, cannot cause the operation to fail. This is usually the result of preconditions being specified as conditions for the operation invocation (ie. promoting the precondition to a higher level in the system hierarchy) or from a system structure which precludes the possibility of the precondition not being met.

a.

```
HOME
        ΔUMACHINE
        projview' = projview * homeproj
        sstart' = (1,1)
        cursor' = (1,1)
        screendoff' = (1,1)
        insertmode' = false
        arrowcommand' = false
        msgline' = 'HOME--',nodoc,'DOCUMENTS IN THE FILING
CABINET'
        show' = homedoc
```

b.

```
CLEAN
        ΔUMACHINE
        projview' = ⟨⟩
        insertmode' = false
        arrowcommand' = false
        msgline' = blankmsg
```

c.

DCREATE

ΔUMACHINE

%%% there must be room for the new document.(ERROR13)

nodoc < maxdocno

%%% there must be room for the new projections.(ERROR8)

novproj + 2 ≤ maxvproj

docdatal' = docdatal
@ {newdoc ↦ emptydoc}
@ {homedoc ↦
docdatal(homedoc) *
⟨blankline,blankline⟩}

%%% show the user a clean new sheet of paper and hide details of the current tabletop.

projview' = projview * ⟨blankproj,nullproj⟩
point' = point ∪ {newdoc ↦ 2 * nodoc + 1}
sstart' = (1,1)
cursor' = (1,1)
screendoff' = (1,1)
insertmode' = false
arrowcommand' = false
msgline' = blankmsg
show' = newdoc

---

d.

DFILE

ΔUMACHINE

---

docdatal' = (docdatal\{newdoc}) ∪
{newname ↦ docdatal (newdoc)}
projview' = delete (projview,novproj-1)
point' = (point\{newdoc}) ∪
{newname ↦ point (newdoc)}

```
      insertmode' = false
      arrowcommand' = false
      msgline' = blankmsg
      show' = tabletop

%%% select a legal but unused name.
      where newname ∈ τ(docname - dom (docdatal))
```

---

e.

DTHROW

```
      ΔUMACHINE
      d              :        docname
```

```
%%% certain documents must remain permanently in the
filing cabinet.
      d ∉ {homedoc, blankdoc}

      docdatal' = docdatal\{d} ⊕
                  {homedoc ↦
                      delete(delete(docdatal(d),point(d)),
                             point(d))}

%%% the corresponding projections must also be deleted.
      projbuild' = deletepb (projbuild) (d)
      projview' = deletepv (projview) (d)
      point' = point\{d}
      insertmode' = false
      arrowcommand' = false
      msgline' = blankmsg
```

---

f.

PROJHERE

ΔUMACHINE

```
%%% the cursor must point to a window (ERROR1) but not a
window on the homedoc or the blankdoc.
      vpno ≠ 0
      NOT innameline(projview,vpno,cursor,sstart)
      dn (masterspot (z)) ∉ {homedoc,blankdoc}

      proj' = (dn(masterspot(z)),retv-vprojection(z)
                        (cursor+sstart-(1,1))
      insertmode' = false
      arrowcommand' = false
      msgline' = blankmsg

where z ≙ projview (vpno)
```

g.

CAMHERE

ΔUMACHINE

```
%%% the cursor must be in a window.(ERROR1)
      vpno ≠ 0

      loc (cam') = (dn(masterspot(a)),retv-vprojection(a)
                              (startpane))
      size (cam') = panesize (a)
      insertmode' = false
      arrowcommand' = false
      msgline' = blankmsg

where a ≙ projview (vpno)
```

h.

PROJCREATE

ΔUMACHINE

%%% there must be room for a new building.
projection.(ERROR8)
nobproj < maxbproj

%%% both the camera area (ERROR14) and the projector area (ERROR15) must still exist.
aread(coord(proj),size(cam)) ⊆
area((1,1),(cols(a),rows(a)))

aread(coord(loc(cam)),size(cam)) ⊆
area((1,1),(cols(b),rows(b)))

projbuild' = projbuild ∪
{nobproj+1 ↦
createproj(loc(cam),size(cam),proj)}
insertmode' = false
arrowcommand' = false
msgline' = blankmsg

where
a ≙ docdata1 (dn (proj))
b ≙ docdata1 (dn (loc (cam)))

---

i.

PROJREMOVE

ΔUMACHINE

%%% this selected spot points to the slave of a building projection.(ERROR16)
bpno ≠ 0

```
      projbuild' = delete(projbuild,bpno)
      insertmode' = false
      arrowcommand' = false
      msgline' = blankmsg
```

---

j.

PROJSWAP

```
      ΔUMACHINE
```

```
%%% two viewing projection slaves do exist at this point
on the tabletop.(ERROR17)
      vpno ≠ 0
      svpno ≠ 0

      projview' = swap-projection (projview,vpno,svpno)
      insertmode' = false
      arrowcommand' = false
      msgline' = blankmsg
```

---

k.

REMOVEWINDOW

```
      ΔUMACHINE
```

```
%%% must be in a window.(ERROR1)
      vpno ≠ 0

      projview' = delete (projview,vpno)
      insertmode' = false
      arrowcommand' = false
      msgline' = blankmsg
```

---

l.

REMOVESG

ΔUMACHINE

% % must be in a window.(ERROR1)
vpno ≠ 0

projview' = projview ⊕
{vpno ↦ removesg(projview(vpno))}
insertmode' = false
arrowcommand' = false
msgline' = blankmsg

---

m.

CREATESG

ΔUMACHINE

%%% must be in a window.(ERROR1)
vpno ≠ 0

%%% the designated pane area must be within the window(ERROR4) if the entire window is not stained glass.
area(sstart,cursor+sstart-(1,1)) ⊆
aread(slavespot(a),projsize(a)) ∨
innameline(projview,vpno,cursor,sstart)

%%% the window currently does not have a pane.(ERROR7)
panesize (a) = (0,0)

NOT innameline(projview,vpno,cursor,sstart) ⇒
projview' = projview ⊕
{vpno ↦
createsg (a,sstart,cursor+sstart-(1,1))}

```
      innameline(projview,vpno,cursor,sstart) ⇒
          projview' = projview ⊕
              {vpno ↦
               createsg
(a,slaves(a),slaves(a)+projsize(a)-(1,1))}

      insertmode' = false
      arrowcommand' = false
      msgline' = blankmsg

      where a ≜ projview (vpno)
```

---

n.

MOVEWINDOW

```
      ΔUMACHINE
      offset      :       coordinate
```

```
%%% must be in a window.(ERROR1)
      vpno ≠ 0

%%%  after a move the window must still be on the tabletop
with room for the nameline.(ERROR5)
      aread(slavespot(a)+offset,projsize(a)) ⊆
                aread((1,2),(tablewidth,tablelength))

%%% the policy on disclosure affects what the user sees as
a result of this operation.
      tabletopdisplay(cursor,sstart,offset,screendoff,b,vpno)

      insertmode' = false
      arrowcommand' = true
      msgline' = 'MOVE WINDOW--USE ARROWS'

      where a ≜ projview (vpno)
            b ≜ projview ⊕
                  {vpno ↦ movewindow (a)(offset)}
```

---

o.

MOVESG

ΔUMACHINE

offset : coordinate

%%% must be in a window.(ERROR1)

vpno ≠ 0

%%% the window has a stained glass.(ERROR2)

panesize (a) ≥ (1,1)

%%% the location of the stained glass after the move must be in the window.(ERROR4)

aread(startpane(a)+offset,panesize(a)) ⊆
aread(slavespot(a),projsize(a))

projview' = projview @
{vpno ↦ movesg (a)(offset)}
insertmode' = false
arrowcommand' = true
msgline' = 'MOVE STAINED GLASS--USE ARROWS'

where a ≔ projview (vpno)

---

p.

SIZEWINDOW

ΔUMACHINE

down,

up,

left,

right, : *INT*

%%% must be in window.(ERROR1)

vpno ≠ 0

```
%%% master stays on document.(ERROR3)
      aread(coord(masterspot(zz)),projsize(zz)) ⊆
         area((1,1),(cols(docdatal(dn(masterspot(zz)))),
                     rows(docdatal(dn(masterspot(zz))))))

%%% window stays on the tabletop with room for the
nameline.(ERROR5)
      aread(slavespot(zz),projsize(zz)) ⊆
         aread((1,2),(tablewidth,tablelength))

%%% the nameline remains visible.(ERROR18)
      projsize (zz) ≥ (1,0)

%%% the policy on disclosure affects what the user sees as
a result of this operation if the offset is not (0,0).
      tabletopdisplay(sstart,cursor,(a,b),screendoff,z,vpno)

      insertmode' = false
      arrowcommand' = true
      msgline' = 'CHANGE WINDOW SIZE--USE ARROWS'

%%% here we define the offset to be used to move the
cursor -- no movement is done unless required to keep
cursor on new window or nameline.
where c ≙ cursor+sstart-(1,1) -
                     slavespot(projview(vpno)) + (0,1)
      d ≙ slavespot(projview(vpno)) +
                     projsize(projview(vpno)) - (1,1) -
                     (cursor+sstart-(1,1))

      left + x(c) < 0 ⇒ a = -(left + x(c))
      right + x(d)  < 0 ⇒ a = right + x(d)
      left + x(c) ≥ 0 ∧ right + x(d) ≥ 0 ⇒ a = 0

      up + y(c)   < 0 ⇒ b = -(up + y(c))
      down + y(d) < 0 ⇒ b = down + y(d)
      up + y(c) ≥ 0 ∧ down + y(d) ≥ 0 ⇒ b = 0
```

```
      z = projview @ {vpno ↦
          sizewindow (projview(vpno))(down,up,right,left)}
      zz = z (vpno)
```

This window sizing operation can be used for both expansion and retraction.

q.

SIZESG

```
      ΔUMACHINE
      up,
      down,
      left,
      right        :        INT
```

```
%%% must be in a window.(ERROR1)
      vpno ≠ 0

%%% the revised stained glass must stay on the
window.(ERROR4)
      aread(startpane(z),panesize(z)) ∈
                  aread(slavespot(z),projsize(z))

%%% the window has a stained glass pane.(ERROR2)
      panesize (projview (vpno)) > (1,1)

%%% the resized pane must still be visible.(ERROR19)
      panesize(z) > (1,1)

      projview' = projview @ {vpno ↦
           sizesg (projview(vpno))(up,down,left,right)}
      insertmode' = false
      arrowcommand' = true
      msgline' = 'CHANGE STAINED GLASS SIZE--USE ARROWS'

      where z = projview' (vpno)
```

---

r.

OFFHOME

ΔUMACHINE

```
projview' = delete (projview,novproj)
sstart' = (1,1)
cursor' = (1,1)
screendoff' = (1,1)
insertmode' = false
arrowcommand' = false
msgline' = blankmsg
show' = tabletop
```

If the user is viewing the home document and wishes to return to the normal tabletop picture he will push the HOME key and activate this operation.

s.

CHOOSE

ΔUMACHINE

```
%%% the user must be viewing the homedoc to select this
operation.
      show = homedoc

%%% we make the decision to preclude the user selected
windows from displaying the homedoc.
      a ≠ homedoc

      projview' = delete (projview,novproj) *
                  < PROJCLASSV [ slavespot/(1,2)
                                 masterspot/(a, (1,1))
                                 projsize/
                                     (cols(a),rows(a))
                                 startpane/(1,2)
                                 panesize/(0,0) ] >
```

```
        sstart' = (1,1)
        cursor' = (1,1)
        screendoff' = (1,1)
        insertmode' = false
        arrowcommand' = false
        msgline' = blankmsg
        show' = tabletop

where a ≜ point^-1 (oddit (y (cursor+sstart-(1,1))))
```

This operation will create a full document sized projection from the document the user has chosen.

t.

SCROLL

```
        ΔUMACHINE
        offset      :       coordinate
```

```
%%% must be in a window.(ERROR1)
        vpno ≠ 0

%%%  the master of the window creating projection must
still be on the master document after this
command.(ERROR3)
        aread(coord(masterspot(a))+offset,projsize(a)) ∈
                area((1,1),(cols(b),rows(b)))

        projview' = projview ⊕
                    {vpno ↦ scrolldoc (a,offset)}
        insertmode' = false
        arrowcommand' = true
        msgline' = 'SCROLL OR ADJUST--USE ARROWS'

where a ≜ projview (vpno)
      b ≜ docdata1 (dn (masterspot (a)))
```

---

u.

INSERTMODE

ΔUMACHINE

---

insertmode' = true
arrowcommand' = false
msgline' = 'INSERTING'

---

v.

INSERT

ΔUMACHINE
s : speck

%%% must be in a window.(ERROR1)
vpno ≠ 0

%%% must be on a line free of building projection slaves or else in the nameline.(ERROR6)
(whichdoc = none) ⇒
innameline(projview,vpno,cursor,sstart)

(whichdoc ≠ none) ⇒
lineok(whichdoc,cols(docdatal(whichdoc)),
y(position),projbuild)

%%% there must be room within the left margin set by window for inserted character.(ERROR12)
endline (y(c),docdatal(d)) ≤
x(coord(masterspot(a)) + projsize(a) - (1,1))

%%% if the cursor is not at margin (right edge of window) then this operation may require disclosure.
x(cursor + sstart - (1,1)) <
x(coord(slavespot(a)) + projsize(a) -(1,1)) ⇒
tabletopdisplay(sstart,cursor,(1,0),
screendoff,projview,vpno)

```
      docdatal' = docdatal @
              {d ↦ inserts (docdatal(d),c,s)
                      P(1..cols(docdata(d)))}

      arrowcommand' = false

where a ≙ projview (vpno)
      b ≙ dn (masterspot (a))
      whichdoc ≠ none ⇒ c ≙ position ∧ d ≙ whichdoc
      whichdoc = none ⇒ c ≙ (x(retv-vprojection(a)
                (cursor+sstart-(1,1))),y(point(b)))
                      ∧ d ≙ homedoc
```

---

w.

DELETE

```
      ΔUMACHINE
```

```
%%% must be in a window.(ERROR1)
      vpno ≠ 0

%%% must be on a line free of building projection slaves
or else in the nameline (ERROR6) (but not at end of
nameline(ERROR20)).
      (whichdoc = none) ⇒
            innameline(projview,vpno,cursor,sstart)
         ∧  x(c) < endline (y(c),docdatal(d))

%%% combining two lines cannot disrupt a building
projection.(ERROR6)
      (whichdoc ≠ none) ⇒
            lineok(whichdoc,cols(docdatal(whichdoc)),
                  y(position),projbuild)
%%% there must be something in the next line.(ERROR20)
         ∧  x(c) > endline (y(c),docdatal(d)) ⇒
              (rows(docdatal(d) > y(c) ∧
```

```
%%% the resultant line must fit in the margins.(ERROR12)
                x(c)+endline(y(c)+1,docdatal(d)) -
                    x(coord(masterspot(a))) ≤
                x(coord(masterspot(a))+projsize(a)-(1,1))

%%% character delete.
x(c) < endline (y(c),docdatal(d)) ⇒
      docdatal' = docdatal * {d ↦
                  deletes(docdatal(d),c) * ⟨backcolor⟩}

%%% carriage return deletion changes document size -- may
also affect projections and length indicator.
x(c) ≥ endline (y(c),docdatal(d)) ⇒
         (docdatal' = docdatal * {d ↦
                      deleteel(docdatal(d),c)} * {homedoc
                          ↦ barupdate(point(d)+1,d,
                             docdatal(homedoc),
                             length(docdatal'(d)))} ∧
          projbuild' =
             bprojreduce(projbuild,d,rows(docdatal'(d)) ∧
          projview' =
             vprojreduce(projview,d,rows(docdatal'(d)))

      insertmode' = false
      arrowcommand' = false
      msgline' = blankmsg

where a ∈ projview (vpno)
      b ∈ dn (masterspot (a))
      whichdoc ≠ none ⇒ c ∈ position ∧ d ∈ whichdoc
      whichdoc = none ⇒ c ∈ (x(retv-vprojection(a)
                  (cursor+sstart-(1,0))),point(b))
                        ∧ d ∈ homedoc
```

```
x.
CUT_____________________________________________________________
      ΔUMACHINE
_____________

%%% must be in a window (not in nameline).(ERROR1)
      vpno ≠ 0
      NOT innameline(projview,vpno,cursor,sstart)

%%% the window has a stained glass.(ERROR2)
      panesize (z) > (1,1)

%%% the stained glass selected lines are not touched by a
building projection slave.(ERROR6)
      areaok (whichdoc,cols(docdatal(whichdoc)),a,
                  a+y(panesize(z))-1,projbuild)

%%% changes document size -- may affect projections and
length indicator.
      docdatal' = docdatal @
                  {whichdoc ↦
                    cut(docdatal(whichdoc),a-1,
                          y(panesize(z)))} @
                  {homedoc ↦
                          barupdate(point(whichdoc)+1,
                          whichdoc,docdatal(homedoc),
                          length(docdatal'(whichdoc)))}
      projbuild' = bprojreduce(projbuild,whichdoc,
                          rows(docdatal'(whichdoc)))
      projview' = vprojreduce(projview,whichdoc,
                          rows(docdatal'(whichdoc)))
      insertmode' = false
      arrowcommand' = false
      msgline' = blankmsg

where z ≙ projview (vpno)
      a ≙ y(retv-vprojection (z) (startpane(z)))
_____________________________________________________________
```

y.

ALTER

ΔUMACHINE

s : speck

%%% must be in a window.(ERROR1)

vpno ≠ 0

docdata1' = docdata1 * {d ↦
alter (docdata1(d),c,s}

%%% if the cursor is not at the margin (edge of the window) then disclosure may be required.

x(cursor+sstart-(1,1)) <
x(coord(slavespot(a))+projsize(a)-(1,1) ⇒
tabletopdisplay (sstart,cursor,(1,0),
screendoff,projview,vpno)
arrowcommand' = false
msgline' = blankmsg

where a ≙ projview (vpno)
b ≙ dn (masterspot (a))
whichdoc ≠ none ⇒ c ≙ position ∧ d ≙ whichdoc
whichdoc = none ⇒ c ≙ (x(retv-vprojection(a)
(cursor+sstart-(1,0))),point(b))
∧ d ≙ homedoc

This operation replaces one speck with another.

z.

NEWLINE

ΔUMACHINE

```
%%% in window but not in nameline (ERROR1) and line free
of building projection slaves.(ERROR6)
      NOT innameline(projview,vpno,sursor,sstart)
      vpno ≠ 0
      lineok (whichdoc,position,docdatal(whichdoc))

%%% there must be room in the document.(ERROR11)
      rows(docdatal(whichdoc)) < maxdoclength

%%% the new line must fit within the margins
given.(ERROR12)
      endline(y(position),docdatal(whichdoc)) -
                  x (position) ≤
                      x (sizeproj(projview(vpno))

%%% the change in the document length may affect the
length indicator.
      docdatal' = docdatal @ {whichdoc ↦
                insertel (docdatal(whichdoc),position)} @
                {homedoc ↦ barlength(point(whichdoc)+1,
                    whichdoc,docdatal(homedoc),
                    length(docdatal'(whichdoc)))}

      arrowcommand' = false

%%% disclosure may affect the picture seen by the user.
      tabletopdisplay (sstart,cursor,
           (x(slavespot(z)) - cursor-sstart+(1,1)),1),
            screendoff,a,vpno)

where z ∈ projview (vpno)
      y(slavespot(z)) > y(cursor+sstart-(1,1)) ⇒
         a = projview
      y(slavespot(z)) = y(cursor+sstart-(1,1)) ⇒
         a = projset @ {vpno ↦ sizewindow(z)(0,1,0,0)}
```

This operation creates two lines from one when given a carriage return.

REVERSE

Basis:

Prior: UMACHINE

Comments:

To allow certain of the required user commands to be reversible, we must save the previous state description. The easiest manner in which to accomplish this is simply to have two versions of the state, the current version and the previous version.

State Definition:

REVERSE

UMACHINE

UMACHINE#

The components of the old version of the state are denoted by the component name with the # superscript affixed.

Operations:

ΔREVERSE

REVERSE

REVERSE'

status' : STATUS

---

status' = SUCCESS

a.

UNDO

ΔREVERSE

θUMACHINE' = θUMACHINE*

θUMACHINE*' = □

UNDO is not reversible; therefore, we don't care what happens to the old state components once they are used to reverse the effects of a command.

b.

PASTE

ΔREVERSE

%%% there must be something to paste

panesize (z) > (1,1)

%%% the user must have selected a receiving location within a spot where no building projection has a slave.

vpno ≠ 0

NOT nameline(projview,vpno,cursor,sstart)

linesok (whichdoc,cols(docdatal(whichdoc)),
y(position),projbuild)

%%% the resulting document must not be too long.

rows(docdatal(whichdoc)) + y(panesize(z)) ≤
maxdoclength

%%% both documents must be the same width.

cols (docdatal (whichdoc)) = cols (c)

```
%%% update the document and its length entry in homedoc.
      docdatal' = docdatal @
                  {whichdoc ↦ paste(docdatal(whichdoc),
                       y(position),pastedoc)} @
                  {homedoc ↦ barupdate(point(whichdoc)+1,
                       whichdoc,docdatal(homedoc),
                       length(docdatal'(whichdoc)))}

%%% return the user to the place from which the material
was copied.
      sstart' = sstart*
      cursor' = cursor*
      screendoff' = screendoff*
      insertmode' = false
      msgline' = blankmsg

where z ≜ projview* (vpno*)
      c ≜ docdatal* (dn(masterspot(z)))

%%% The document to be pasted in is created from the donor
area by determining the appearance of each point in the
area.
%%% This means that shared information and the information
basic to the donor document is pasted.
      pastedoc : ARRAY [ speck ]
      length (pastedoc) = y ( panesize (z))
      (∀a : 1..length(pastedoc))
         (length (pastedoc (a)) = cols(c))

(∀x : 1..cols(c)) (∀y : 1..length(pastedoc))
   (pastedoc (y) (x) =
        prjview' ((FILE-APPEARANCE_FILER7A;
                   PROJECTED-APPEARANCE_FILER7A)
                      (c, (x, y(c(masterspot(z)))+y-1))))
```

The paste command uses the previous state description to define where the material to be pasted is located and where to return the user

after the paste command terminates. Notice that since this command pastes in the information as it appears to the user (ie. including copies of shared information), there is the prossibility of attemping to paste undefined specks.

c.

CREATEWINDOW

ΔREVERSE

%%% The following represents the decision to disallow a user defined window from showing the home document.

```
a ≠ homedoc
```

%%% the document must not be too small to fill the entire window.(ERROR10)

```
docdata2 (a) > y(stop-start) + 1

projview' = delete (projview,novproj) *
          < PROJCLASSV[slavespot/start;
                       masterspot/(a,(1,b));
                       projsize/(stop-start+(1,1));
                       startpane/start;
                       panesize/(0,0)] >
sstart' = sstart*
cursor' = cursor*
screendoff' = screendoff*
insertmode' = false
arrowcommand' = false
msgline' = blankmsg
show' = tabletop

where a ≔ point^-1 (oddit (y (cursor+sstart-(1,1))))
      10 * (x(position)-1) + y(stop-start) +1
                           > docdata2(a) ⇒
            b = 10 * x(position)-1) + 1
      10 * (x(position)-1) + y(stop-start) +1
                           <  docdata2(a) ⇒
            b = docdata2(a) - y(stop-start)
```

This operation creates a new window in the location directed by the user. The master of the window is selected by the user at the home document. After this operation the display is returned to the tabletop area which shows the new window.

MACHINE1

Basis:

Prior: REVERSE

Comments:

At this level of specification we can introduce the system hardware representation into our machine state. The selected target machine is the LSI 11-03 with printer (or print file), floppy disk, and KGM PT700 character display screen. This hardware will accommodate all of the prior assumptions we have made about the underlying machine (eg. reverse video, textual display screen, etc.) Because of the availability of disk storage, we will specify that copies of the filing cabinet documents are on secondary storage. That will allow the option of removing the now derived documents from main store. It is unlikely that the machine will be big enough to permit the filing cabinet to be completely main storage resident.

Requirements / Design Decisions:

[[1]] All documents are filled with characters only.

[[2]] The hardware selected is the LSI 11-03 with disk, printer, and textual display terminal. A standard keyboard with directional keys and 28 definable function keys provide the user input.

Auxiliary Definitions:

a. `fname is ABSTRACT`

Disk filename formats are left unspecified.

b. `record is ABSTRACT`

The organization of data on the disk is left unspecified. We will not attempt to define what constitutes a record but will assume that whatever we specify to go on the disk will satisfy the criteria for the set of records.

c.

character ≙ {a,b,c,d,e,f,g,h,i,j,k,l,m,
n,o,p,q,r,s,t,u,v,w,x,y,z,
A,B,C,D,E,F,G,H,I,J,K,L,M,
N,O,P,Q,R,S,T,U,V,W,X,Y,Z,
!,",#,$,%,&,',(,),_,~,|,',
{,+,*,},<,>,?,,,.,/,:,:,],
@,[,-,↑,\,1,2,3,4,5,6,7,8,9,0}
∪ {sp}

This is the standard *visible* ASCII character set which will be used to create the documents.

d.

speck ≙ character ∪
revvideo ((character))

Specks are visible on the screen as well as including the character set used to create documents. Therefore, the specks must include all characters in reverse video.

e.

docdisk ∈ docname → fname

docdisk$^{\#}$ ∈ docname → fname

ran (docdisk) ∩ ran (docdisk$^{\#}$) = Φ

These two functions turn document names into their corresponding disk file names. The decorated function will be used to give unique disk file names to documents of the previous state.

f.

DISK

fname ⇸ SEQ [record]

This schema allows us to describe the system disk very abstractly.

State Definition:

MACHINE1

[[2]]

%%% secondary store

disk : DISK

%%% printer

printer : SEQ [line : SEQ [character]]

%%% main storage

[[1]] REVERSE (DOCUMENTCLASS[speck] ⤔ DOCUMENTCLASS[character])

docdata2 : docname ⇸ ℕ

%%% the lengths of the documents will be a useful derived component to define.

dom(docdata1) = dom(docdata2)

(∀d : dom(docdata1))
(docdata2(d) = rows(docdata1(d))

%%% both the old and the current documents are on the disk and must have unique disk names.

(∀d : dom (docdata1$^{\#}$))
(disk (docdisk$^{\#}$(d)) = docdata1$^{\#}$ (d))

(∀d : dom (docdata1))
(disk (docdisk (d)) = docdata1 (d))

Operations:

a.

PRINT

ΔMACHINE1

---

printer' = printer * P
insertmode' = false
arrowcommand' = false
msgline' = blankmsg

where
%%% Print the whole document as it exists in the filing cabinet.
(show = homedoc) ⇒ P = docdata1(a)
a ≙ point^{-1} (oddit (y (cursor+sstart-(1,1))))

%%% Print the screen.
(show ≠ homedoc ∧
NOT innameline(projview,vpno,cursor,sstart)) ⇒
(P ∈ ARRAY [character] ∧
length(P) = screenlength ∧
(∀a : 1..screenlength)
(length(P(a)) = screenwidth) ∧
(∀x : 1..screenwidth)(∀y : 1..screenlength)
(P(y)(x) = screendoc
(y(sstart)+y-y(screendoc))

(x(sstart)+x-x(screendoc))))

```
%%% Print the entire window.
      (show ≠ homedoc ∧
         innameline(projview,vpno,cursor,sstart)) ⇒
           (P ∈ ARRAY [ character ] ∧
            length(P) = y (projsize (z)) ∧
            (∀a : 1..length(P))
                (length(P(a)) = x (projsize (z))) ∧
            (∀x : 1..x(projsize(z)))(∀y : 1..length(P))
                (P(y)(x) = prjview'
                              ((FILE-APPEARANCE_FILER7A;
                                PROJECTED-APPEARANCE_FILER7A)
                                 (dn(masterspot(z),
                                 c(masterspot(z))-(1,1)))))
      where z ≙ projview (vpno)
```

The PRINT command prints the entire stored document, the screen, or the window as determined by the location of the cursor.

MACHINE1A

Basis:

Prior: MACHINE1

Comments:

Only some of the user commands change documents in the filing cabinet and even these commands change only a few documents. It is unnecessary, therefore, to save the entire set of documents that were in the old filing cabinet just in case the user changes his mind with a reverse command. This refinement utilizes that fact to redefine the disk in a more economic fashion by not keeping duplicates of unchanged documents.

State Definition:

MACHINE1A

MACHINE1

docname1	:	edocname
docname2	:	edocname
docname3	:	edocname
disk1	:	DISK

$disk1 = disk \upharpoonright a$

$docname1 = none \Rightarrow docname2 = none$

$docname2 = none \Rightarrow docname3 = none$

where $a \triangleq docdisk \langle\!\langle dom(point) \rangle\!\rangle \cup docdisk^{\#} \langle\!\langle \{docname1, docname2, docname3\} - \{none\} \rangle\!\rangle$

The most a user command could change is three documents; therefore, we add three document name components to the state to hold the names of the affected documents. The new disk component will hold only the current documents and up to three old documents required in case of a command reversal.

Operations:

ΔMACHINE1A

MACHINE1A
MACHINE1A'

status' : STATUS

---

status' = SUCCESS

---

a.

REV

ΔMACHINE1A
UNDO

---

docname1' = none
docname2' = none
docname3' = none

The reverse removes the backup documents after making them the current documents.

b.

DCREATE

ΔMACHINE1A
$DCREATE_{REVERSE}$

---

docname1' = homedoc
docname2' = none
docname3' = none

The homedoc has been changed in preparation for the creation of a new document.

c.

DFILE

ΔMACHINE1A

$DFILE_{REVERSE}$

docname1' = homedoc

docname2' = newdoc

docname3' = none

Filing a document has deleted the unfinished newdoc and changed the homedoc.

d.

DTHROW

ΔMACHINE1A

$DTHROW_{REVERSE}$

docname1' = homedoc

docname2' = d

docname3' = none

Throwing a document away affects the homedoc and the document removed.

e.

PASTE

ΔMACHINE1A

$PASTE_{REVERSE}$

docname1' = whichdoc

docname2' = homedoc

docname3' = none

Pasting changes the document pasted into and may change document length infomation.

f.

CUT

ΔMACHINE1A

$CUT_{REVERSE}$

docname1' = whichdoc
docname2' = homedoc
docname3' = none

In cutting part out of a document, the length of the document may change.

g.

PNC

ΔMACHINE1A

$PASTE_{REVERSE}$;$CUT_{REVERSE}$
docname1' = whichdoc
docname2' = whichdoc*
docname3' = none

This operation changes the documents cut from and pasted to, as well as the homedoc since the document lengths may have changed.

MACHINE2

Basis:

Prior: MACHINE1A

Comments:

In this final level, we have completely recomposed the system components and have developed a total system (ie. it will now process all possible (legal and illegal) user commands). We also developed here the system control structure which interfaces with the user and controls the activity of the system. The three standard system phases -- initialization, operation, and termination -- have been defined.

The provisional user's manual was taken as final authority on the functioning of the system in general and on the operation of the user commands. Other decisions reflected in this level of specification include the final determination of document paper size and the selection of key inputs over a menu system. The state space has been updated to facilitate system control by maintaining a short *history* of key inputs.

Requirements / Design Decisions:

[[1]] All document papers will be exactly the same as the the width of the screen to simplify document manipulation. Of course, documents can be made at any width up to the document paper width. Selecting a document size larger than the screen width would have made document viewing very difficult on a fixed width screen with doubtful improvement in the system usefulness or generality.

[[2]] Key inputs rather than menu selection will be used.

[[3]] The decisions about overall system function and user command functions are as indicated in the provisional user's manual unless a change is noted at the appropriate point in the specification.

Auxiliary Definitions:

a.

`arrowkeys = {left, right, up, down, tab}`

These are the keys which move the cursor or give directional information to the commands which require it.

b.

legalkeys ≜ arrowkeys ∪
character ∪ {el} ∪
{khome, kdthrow, kdfile, kdcreate, kinsert,
kdelete, kpaste, kcut, kpnc, kprint, kwremove,
ksgremove, kwcreate, ksgcreate, kwmove,
ksgmove, kwretract, ksgretract, kwexpand,
ksgexpand, kscroll, kswap, kchoose, kpcreate,
kpremove, kcam, kproj, kclean, kquit,
krev, kaccept}
legalkeys ⊆ possiblekeys

Here we have defined the subset of possible keys which are valid in this system.

c.

screenkeys ≜ legalkeys - {kdfile, kchoose}

Screenkeys defines the set of user keys which can be selected when the user display shows the tabletop.

d.

newkeys ≜ legalkeys - {khome, kdcreate, kscroll, kswap,
kchoose, kpcreate, kpremove, kcam,
kproj, kclean, kquit, kwremove,
kwcreate, kwmove}

These keys are valid when the user is creating a new document for the filing cabinet.

e.

homekeys ≜ arrowkeys ∪ {kdthrow, kprint, kchoose, kclean,
khome, kquit, krev}

This is the set of valid keys available if the screen is displaying the home document.

f.

```
reversekeys = legalkeys - characters - arrowkeys
              - {el, kinsert, kdelete, kprint, kquit,
                 krev, kaccept}
```

Some commands are not reversible. The set which are reversible is defined by reversekeys.

```
g.    maxdocwidth = 80
      screenlength = 23
      screenwidth = 80
      backcolor = sp
      marker = '-'
```

These system constants can now be defined based on known desires of the user or limitations of the selected hardware.

h.

```
screendisplay is ABSTRACT.
```

The policy for arrow movement of the cursor is left undefined at this point but will specify how the cursor, screen, and screen document react to movement of the cursor on the screen.

State Definition:

```
MACHINE2_____________________________________________
      MACHINE1A (DOCUMENTCLASS ⊢⊲
                 DOCUMENTCLASS (cols = maxdocwidth))
      newkey      :       possiblekeys
      oldkey      :       possiblekeys
_____________________________________________________
```

The state definition is updated to include the most recent previous reversible user command and the current user command. The old key will be useful for reversing and for operations which are dependent upon the history of the session. Now the state will also require all documents to be the same width as the screen.

Operations:

a.

SESSION

ΔDISK

θDISK' = INITIALIZE (θDISK);
EXECUTE;
TERMINATE

A session of the windowing screen editor includes the standard three parts for a non-continuous system. The environment which exists between the sessions of the system is modeled by the DISK.

b.

INITIALIZE

d : DISK
m' : MACHINE2

---

projbuild' = d ('projbuild')
projview' = d ('projview')
point' = d ('point')
sstart' = first (d ('view'))
proj' = fifth (d ('view'))
cam' = fourth (d ('view'))
cursor' = second (d ('view'))
screendoff' = third (d ('view'))
insertmode' = false
arrowcommand' = false
start' = □
stop' = □
msgline' = blankmsg
θUMACHINE*' = □
printer' = ⟨⟩
docname1' = none
docname2' = none
docname3' = none

newkey' = □
oldkey' = krev

The initial state representation is generally the same as the final state representation of the previous session. However, no history (previous user commands) is remembered from the last session.

c.

EXECUTE ≙ $DISPLAYED_{MACHINE2}$; USER;
LOOP (OPERATE; $DISPLAYED_{MACHINE2}$; USER)

The execution of the user's commands includes the display of the screen to the user and the input of user command desires.

d.

USER

ΦMACHINE1A
newkey,
newkey',
oldkey,
oldkey' : possiblekeys

status' : STATUS

---

show = tabletop ⇒ newkey' = τ(screenkeys)
show = newdoc ⇒ newkey' = τ(newkeys)
show = homedoc ⇒ newkey' = τ(homekeys)
oldkey ∈ {kpaste,kpnc} ⇒
newkey' = τ(arrowkeys ∪ {kaccept})
∨ NOT arrowcommand
oldkey = kwcreate ⇒
newkey' = τ({kquit,krev,kchoose} ∪
arrowkeys)
oldkey' = oldkey

status' = SUCCESS

Only certain input commands are allowed by the system, depending upon what is being displayed to the user at the time of command input.

BADKEY?

```
ΦMACHINE1A
newkey,
newkey',
oldkey,
oldkey'      :       possiblekeys

status'      :       STATUS
```

```
show = tabletop ⇒
       newkey' = τ(possiblekeys-screenkeys)
show = newdoc   ⇒
       newkey' = τ(possiblekeys-newkeys)
show = homekey  ⇒
       newkey' = τ(possiblekeys-homekeys)
oldkey ∈ {kpaste,kpnc} ⇒
         newkey' = τ(possiblekeys-arrowkeys
                                  -{kaccept})
                 ∧ arrowcommand
oldkey = kwcreate ⇒
         newkey' = τ(possiblekeys-
                   {kquit,krev,kchoose}-arrowkeys)
oldkey' = oldkey

status' = ERROR23
```

This error condition traps keys which are not allowed due to the current condition of the display. Consequently, the total user can be modeled by the following:

```
USER ≙ LOOP (BADKEY?); USER
```

e.

OPERATE ≙ (KEYUPDATE ∨ ∥(MACHINE2 | newkey ≠ kquit));
COMMAND

The operation of the user input command is done in two parts -- the updating of information required for command reversing (if new command is reversible) and the actual command execution.

f.

KEYUPDATE

ΔMACHINE1A
newkey,
newkey',
oldkey,
oldkey' : possiblekeys

---

newkey ∈ reversekey

θUMACHINE*' = θUMACHINE
oldkey' = newkey
docname1' = none
docname2' = none
docname3' = none
disk1' = disk1 ⁋ ran (docdisk)

Each time a new reversible key is accepted as a potentially legal command, the current state of the system must be saved so that the system can be returned to that state if a reverse is directed. If the input command is not reversible, then the state before the most recent previous reversible command (if any) is kept.

g.

COMMAND

ΔMACHINE1A
newkey,
newkey',
oldkey,
oldkey' : possiblekeys

status' : STATUS

---

newkey ≠ kquit
(newkey = krev) ⇒ (oldkey' = krev)

(newkey = khome ∧ show = tabletop) ⇒ $HOME_{MACHINE1A}$

(newkey = khome ∧ show = homedoc) ⇒ $OFFHOME_{MACHINE1A}$

(newkey = kdthrow ∧ show = newdoc) ⇒
$DTHROW_{MACHINE1A}$ [d/newdoc];
$REMOVEWINDOW_{MACHINE1A}$; $HOME_{MACHINE1A}$

(newkey = kdthrow ∧ show = homedoc) ⇒
$DTHROW_{MACHINE1A}$ [d/a]

(newkey = kdthrow ∧ show = tabletop) ⇒
$DTHROW_{MACHINE1A}$ [d/dn(masterspot(projview(vpno)))]

(newkey = kdfile) ⇒ $DFILE_{MACHINE1A}$

(newkey = kdcreate) ⇒ $DCREATE_{MACHINE1A}$

(newkey = kinsert) ⇒ $INSERTMODE_{MACHINE1A}$

(newkey = kdelete) ⇒ $DELETE_{MACHINE1A}$

(newkey = kpaste) ⇒ PASTEIT

(newkey = kcut) $\Rightarrow$ $CUT_{MACHINE1A}$

(newkey = kpnc) $\Rightarrow$ PASTEIT

(newkey = kprint) $\Rightarrow$ $PRINT_{MACHINE1A}$

(newkey = kwremove) $\Rightarrow$ $REMOVEWINDOW_{MACHINE1A}$

(newkey = ksgremove) $\Rightarrow$ $REMOVESG_{MACHINE1A}$

(newkey = kwcreate) $\Rightarrow$ WCREATE; $HOME_{MACHINE1A}$

(newkey = ksgcreate) $\Rightarrow$ $CREATESG_{MACHINE1A}$

(newkey = kwmove) $\Rightarrow$
$MOVEWINDOW_{MACHINE1A}$ [offset/(0,0)]

(newkey = ksgmove) $\Rightarrow$
$MOVESG_{MACHINE1A}$ [offset/(0,0)]

(newkey = kwretract) $\Rightarrow$
$SIZEWINDOW_{MACHINE1A}$ [up/0; down/0; left/0; right/0]

(newkey = ksgretract) $\Rightarrow$
$SIZESG_{MACHINE1A}$ [up/0; down/0; left/0; right/0]

(newkey = kwexpand) $\Rightarrow$
$SIZEWINDOW_{MACHINE1A}$ [up/0; down/0; left/0; right/0]

(newkey = ksgexpand) $\Rightarrow$
$SIZESG_{MACHINE1A}$ [up/0; down/0; left/0; right/0]

(newkey = kscroll) $\Rightarrow$ $SCROLL_{MACHINE1A}$ [offset/(0,0)]

(newkey = kchoose $\wedge$ oldkey = kwcreate) $\Rightarrow$ CREATEWINDOW

(newkey = kchoose $\wedge$ oldkey $\neq$ kwcreate) $\Rightarrow$ $CHOOSE_{MACHINE1A}$

(newkey = kpcreate) $\Rightarrow$ $\text{PROJCREATE}_{\text{MACHINE1A}}$

(newkey = kpremove) $\Rightarrow$ $\text{PROJREMOVE}_{\text{MACHINE1A}}$

(newkey = kswap) $\Rightarrow$ $\text{PROJSWAP}_{\text{MACHINE1A}}$

(newkey = kcam) $\Rightarrow$ $\text{CAMHERE}_{\text{MACHINE1A}}$

(newkey = kproj) $\Rightarrow$ $\text{PROJHERE}_{\text{MACHINE1A}}$

(newkey = kclean) $\Rightarrow$ $\text{CLEAN}_{\text{MACHINE1A}}$; $\text{HOME}_{\text{MACHINE1A}}$

(newkey = krev $\wedge$ oldkey ϵ reversekey) $\Rightarrow$ $\text{REV}_{\text{MACHINE1A}}$

(newkey = kaccept) $\Rightarrow$ ACCEPT

(newkey ϵ arrowkeys) $\Rightarrow$ CURSORMOVE

(newkey ϵ character) $\wedge$ insertmode $\Rightarrow$
$\text{INSERT}_{\text{MACHINE1A}}$ [char/newkey]

(newkey ϵ character) $\wedge$ NOT insertmode $\Rightarrow$
$\text{ALTER}_{\text{MACHINE1A}}$ [char/newkey]

(newkey = el) $\Rightarrow$ $\text{NEWLINE}_{\text{MACHINE1A}}$

where a = point^{-1} (oddit (y (cursor+sstart-(1,1))))

When the command from the user is other than to terminate the session, this operation will attend to the command. The only error condition not handled by the individual key operations is the case where there is nothing to reverse and the user has directed that a reverse be done. This is taken care of by the following error condition:

NOTHINGTOREVERSE?

ΔMACHINE1A

newkey,

newkey',

oldkey,

oldkey' : possiblekeys

status' : STATUS

---

newkey = krev

oldkey ∉ reversekey

status' = ERROR9

---

h.

TERMINATE

m : MACHINE1A

newkey : possiblekeys

oldkey : possiblekeys

d' : DISK

---

newkey = kquit

d' = disk1 ⩤ ran (docdisk) ⦲

{'projview' ↦ projview,
'projbuild' ↦ projbuild,
'point' ↦ point,
'view' ↦ ⟨sstart,cursor,
screendoff,cam,proj⟩

If the user has commanded the termination of the system then this operation will complete the session by saving the important parts of the state in the environment.

i.

CURSORMOVE

ΔMACHINE1A

newkey,

oldkey : legalkeys

---

arrowcommand ∧

((oldkey = kwmove) ⇒

MOVEWINDOW$_{MACHINE1A}$ [offset/(a,b)]

arrowcommand ∧

(oldkey = ksgmove) ⇒

MOVESG$_{MACHINE1A}$ [offset/(a,b)]

arrowcommand ∧

(oldkey = kwexpand) ⇒

SIZEWINDOW$_{MACHINE1A}$

[up/c; down/d; left/e; right/f]

arrowcommand ∧

(oldkey = kwretract) ⇒

SIZEWINDOW$_{MACHINE1A}$

[up/g; down/h; left/i; right/j]

arrowcommand ∧

(oldkey = ksgexpand) ⇒

SIZESG$_{MACHINE1A}$

[up/c; down/d; left/e; right/f]

arrowcommand ∧

(oldkey = ksgretract) ⇒

SIZESG$_{MACHINE1A}$

[up/g; down/h; left/i; right/j]

arrowcommand ∧

(oldkey = kscroll) ⇒

SCROLL$_{MACHINE1A}$ [offset/(a,b)])

```
(oldkey ∉
      {kwmove,ksgmove,kwexpand,ksgexpand,
       kwretract,ksgretract,kscroll} ∨
         NOT arrowcommand)
      ∧
        %%% the cursor must stay on the window.(ERROR21)
        ((show ∈ {newdoc,homedoc} ∧
          sstart+cursor-(1,1)+(a,b) ∈
              aread(slavespot(z)-(0,1),projsize(z)+(0,1)))
         ∨
          %%% the cursor must stay on the tabletop
area.(ERROR22)
          (show = tabletop ∧
           sstart+cursor-(1,1)+(a,b) ∈
              area((1,1),(tablewidth,tablelength)))) ⇒

%%% the policy for movement of the cursor, screen, and
screendoc has not yet been defined.
      (screendisplay(cursor,sstart,(a,b),screendoff)
       insertmode' = false
       oldkey ∉ {kpaste,kpnc} ⇒
           msgline' = blankmsg
       oldkey ∈ {kpaste,kpnc} ⇒
           msgline' = 'PASTING--POSITION WITH ARROWS, END
WITH ACCEPT')

where
(newkey = left) ⇒ a = -1 ∧ b = 0
(newkey = right) ⇒ a = +1 ∧ b = 0
(newkey = up) ⇒ a = 0 ∧ b = -1
(newkey = down) ⇒ a = 0 ∧ b = +1
(newkey = tab) ⇒ a = 5 ∧ b = 0

(newkey = left) ⇒ e = 1 ∧ c = d = f = 0
(newkey = right) ⇒ f = 1 ∧ c = d = e = 0
(newkey = up) ⇒ c = 1 ∧ d = e = f = 0
(newkey = down) ⇒ d = 1 ∧ c = e = f = 0
(newkey = tab) ⇒ f = 5 ∧ c = d = e = 0
```

(newkey = right) ⇒ i = -1 ∧ g = h = j = 0
(newkey = left) ⇒ j = -1 ∧ g = h = i = 0
(newkey = down) ⇒ g = -1 ∧ h = i = j = 0
(newkey = up) ⇒ h = -1 ∧ g = i = j = 0
(newkey = tab) ⇒ i = -5 ∧ g = h = j = 0

z ≙ projview (novproj)

This operation establishes (albeit abstractly) the effect on the display of *moving* the cursor with the directional commands. Notice that in some cases the use of the directional commands will give directional parameters to other operations rather than directly moving the cursor.

j.

ACCEPT

ΔMACHINE1A
newkey,
oldkey : legalkeys

%%% the cursor must point to a window.(ERROR1)
newkey ∈ {kpaste,kpnc} ⇒ (vpno ≠ 0 ∧
NOT innameline(projview,vpno,cursor,sstart) ∧
%%% the document cannot become too long.(ERROR11)
rows(a) + y(panesize(b)) ≤ maxdoclength ∧
%%% the lines to be split by the paste must not contain a building projection which would be disturbed.(ERROR6)
linesok(whichdoc,cols(docdata1(whichdoc)),
y(postion),projbuild))

(newkey = kpaste) ⇒ $PASTE_{MACHINE1A}$

(newkey = kpnc) ⇒ $PNC_{MACHINE1A}$

arrowcommand' = false

where

a ≙ dn (masterspot (projview (vpno)))

b ≙ projview* (vpno*)

This is a very safe operation which will do nothing except to end a paste or paste and cut operation or stop other arrowcommand operations.

k.

PASTEIT

ΔMACHINE1A

newkey : legalkeys

%%% the cursor must be in a window.(ERROR1)

vpno ≠ 0

%%% the stained glass pane must exist in this window.(ERROR2)

panesize (projview (vpno)) ≥ (1,1)

%%% in the case of paste and cut, the cursor cannot be in the nameline(ERROR1) and the area to cut must not include shared information.(ERROR6)

(newkey = kpnc) ⇒ whichdoc ≠ none ∧
areaok(whichdoc,cols(docdata1(whichdoc)),
a,a+y(panesize(z))-1,projbuild)

insertmode' = false

arrowcommand' = true

msgline' = 'PASTING--POSITION WITH ARROWS, END WITH ACCEPT'

where z = projview (vpno)

a = y(retv-vprojection (z) (startpane(z)))

This is the set up command for a paste or paste and cut operation. The user selects the area to be copied (or cut) with this command and concludes the pasting of the selected information with the ACCEPT.

1.

WCREATE

ΔMACHINE1A

%%% the selected window location must leave room for the nameline on the tabletop.(ERROR5)

area(sstart,sstart+cursor-(1,1)) ⊆
area((1,2),(tablewidth,tablelength))

%%% this must not exceed the limit on number of allowable viewing documents.(ERROR8)

novproj < maxvproj

start' = sstart
stop' = sstart + cursor - (1,1)
insertmode' = false
arrowcommand' = false
msgline' = 'CHOOSE A DOCUMENT TO PROJECT, PLEASE'

This is a set up operation for the creation of new window. With this operation the user has selected the window location and must now select a document to place in the new window.

Status Values

```
ERROR1 ← 'THE CURSOR IS NOT IN A WINDOW'
ERROR2 ← 'A STAINED GLASS PANE DOES NOT EXIST IN THIS
WINDOW'
ERROR3 ← 'THE MASTER OF THIS PROJECTION MUST STAY ON THE
DOCUMENT'
ERROR4 ← 'THE STAINED GLASS PANE MUST STAY ON THE WINDOW'
ERROR5 ← 'THE WINDOW (AND NAMELINE) MUST STAY ON THE
TABLETOP'
ERROR6 ← 'THIS EDIT COMMAND WOULD DISRUPT SHARED
INFORMATION'
ERROR7 ← 'THIS WINDOW ALREADY HAS A STAINED GLASS PANE'
ERROR8 ← 'THERE IS NO ROOM FOR THE REQUIRED NEW PROJECTION'
ERROR9 ← 'THERE IS NO COMMAND TO REVERSE AT THIS TIME'
ERROR10 ← 'THE WINDOW IS TOO LARGE FOR THE INTENDED
DOCUMENT'
ERROR11 ← 'THE DOCUMENT CANNOT EXTEND BEYOND ITS ALLOWED
LENGTH'
ERROR12 ← 'THE UPDATED LINE WOULD NOT FIT WITHIN THE CURRENT
MARGINS'
ERROR13 ← 'THERE IS NO ROOM FOR A NEW DOCUMENT IN THE FILING
CABINET'
ERROR14 ← 'THE DESIGNATED PROJECTOR AREA DOES NOT CURRENTLY
EXIST'
ERROR15 ← 'THE DESIGNATED CAMERA AREA DOES NOT CURRENTLY
EXIST'
ERROR16 ← 'THE INDICATED SPOT HAS NO SHARED DATA TO REMOVE'
ERROR17 ← 'TWO WINDOWS ARE NOT OVERLAPPING AT THIS POINT'
ERROR18 ← 'THE NAMELINE MUST NOT BECOME INVISIBLE'
ERROR19 ← 'THE S. G. PANE CANNOT BE REDUCED ANY FURTHER IN
THIS DIRECTION'
ERROR20 ← 'THERE IS NOTHING TO DELETE'
ERROR21 ← BUZZ
ERROR22 ← BUZZ
ERROR23 ← BUZZ
```

V.2 Implementation Plan.

There are two related areas of the design which have not been specified -- the display development and the disclosure policy. The algorithms for these two segments of the design will be discussed in the first section of this implementation plan. All other required initial prototype algorithms are immediate or obvious from the design specification. The second section will deal with a variety of other significant details about the implementation.

The display and the disclosure policy.

Although the APPEARANCE observation defines the appearance of each point on the tabletop, it does not provide a practical algorithm for the implementation of that abstract definition. A number of possible approaches exist but they can generally be divided into two categories -- 1) screendoc coordinate definition or 2) viewing projection application. The former requires that each coordinate position on the tabletop (or its selected subset, screendoc) be defined individually. The latter suggests initializing the screendoc to the appropriate background color and then *placing* the windows in order on the screendoc. Both algorithms would require the determination of the ultimate master for each coordinate position within a viewing window. Either algorithm could benefit from appropriate *speed-ups*, eg. where possible use rows instead of simple coordinate positions. A very much simplified prototype has shown that on sparsely windowed displays or largely windowed displays with little overlap of windows, the second approach is generally faster. Since a highly cluttered tabletop with many overlapping windows is not a likely mode of utilization for this system, the viewing projection application algorithm appears to be the best choice.

Other decisions required in the definition of the display algorithm include the frequency of screen display refreshment, and the amount of screendoc area redefined before each display refreshment. The simplest starting plan would be to totally rebuild the screendoc and refresh the display after each user input. A significant and almost immediate improvement can be gained by limiting display refreshment to those valid inputs which change the screen. This would eliminate rebuilding the screen and refreshing the display after error inputs, many cursor movement commands, printing, etc.

Of course, total rebuilding of the screendoc before each refreshment is the simplest implementation since it can be centralized and is not dependent upon the type of display change required. With this implementation there is no advantage to having the screendoc larger than the screen area.

This simple base point (a viewing projection application driven screendoc building algorithm with simple *line-at-a-time* speed-ups along with a policy of building and refreshing the entire screen with each screen changing user command) can be altered as necessary to conform with the user's non-functional requirements for the speed of display production if allowed by remaining memory resources (speed vs memory trade off).

The display disclosure policy determines how (if) the screen is rebuilt when the cursor is forced to move by a user command. This happens, for example, during editing operations (insert, alter) to advance the cursor to the next character, during window sizing and movement to maintain the cursor within the current window, or with simple arrow keys designed to move the cursor. Two forms of disclosure are required by these various cursor movement commands -- screendoc disclosure to keep the cursor on the screen and window disclosure to keep the cursor on the current window. Screendoc disclosure is potentially necessary for any user command that moves the cursor. Window disclosure is only required in operations where the cursor is required to remain within the current window and there is the possibility that the cursor may be moved out of this window or onto an overlapping window. The user commands which require this latter type of disclosure are: INSERT, ALTER, NEWLINE, SIZEWINDOW, and MOVEWINDOW.

Screendoc disclosure may require that the screen be moved on the screendoc or the screendoc moved on the tabletop. The policy for screen movement is that the scrolling capability of the video display unit should be utilized where possible. Since left and right scrolling (panning) are not a feature of the target terminal, and therefore full screen refreshment will be required, panning should be implemented to move the screen slightly more than appears immediately necessary in order to minimize the number of required screen refreshments. Whenever the screendoc is also required to move, it should (if possible) have the current screen centered within it.

Implementing window disclosure in operations where the cursor may be forced to follow the window (eg. SIZEWINDOW, MOVEWINDOW) will require the shifting of window projection order (strength). The algorithm for window reordering should not affect the relative strength of any window except the

one to be disclosed (ie. the remainder of the display must be preserved).

The following predicates of the state space and observations are relevant to the implementation of the display development and disclosure policy algorithms:

DISPLAYED ≙ FILE-APPEARANCE; PROJECTED-APPEARANCE;
APPEARANCE; DISPLAY-APPEARANCE

1. cursor ∈ area((1,1), (screenwidth,screenlength))

The cursor stays on the screen.

2. screendoff ∈ area((1,1), (tablewidth-sdwidth+1,
tablelength-sdlength+1))

The screendoc stays on the tabletop.

3. sstart ∈ aread(screendoff, (sdwidth-screenwidth+1,
sdlength-screenlength+1))

The screen stays on the screendoc.

Other implementation considerations:

a. Physical to logical key association.

The lack of adequate user function key space can be solved by using a single key for corresponding pairs of commands which affect either the window or the stained glass pane (ie. move, retract, expand, remove, create.) This will reduce by 5 the number of keys required for user commands. Of course, a second user input would be required to differentiate window from stained glass pane commands.

The placement of the various user key is left to the implementation and will be based on ergonomic considerations.

b. Undefined appearance.

The specified system does not provide a technique to trap undefined coordinate appearance due to the mutual circularity of information sharing projections. The potential result is a non-ending search for an ultimate master location and hence a useless system. The algorithm to detect such problems should determine if the search for an ultimate master ever identifies the same intermediate master location for a second time. This is the practical manifestation of mutual circularity.

An even stronger constraint could be applied to eliminate the possibility of excessively long (but not necessarily circular) searches for an ultimate master. The list of intermediate master locations could be limited, as an arbitrary example, to three times the number of allowable building projections. This would place a worst case limit on the time required to construct the screendoc.

When a location on the screendoc is trapped by the algorithm to detect mutual circularity or limit ultimate master searches, then a symbolic (undefined) character should be taken as the contents of that screendoc location. The character "?" is often used for this purpose.

c. Window appearance.

In addition to the specified use of a reverse video nameline to head a window, some form of border should be used to delineate the remaining three sides. However these border markings should not constitute part of the window information.

d. Non-functional requirements.

Due to the size of the required target machine, there is a very real trade off among speed (eg. user command execution and display generation wait time), the limited main memory space, and capability (eg. the number of projections and documents allowed). The best ultimate balance of these factors is impossible to determine until a fairly complete prototype has been developed for user tests. However, early prototypes ([Richardson,81] and [Richardson,82] have given valuable insight into the appropriate system design for the non-functional requirements. This insight is manifested in the design specification as redundant (derived) components which give greater flexibility to the implementation. It also allows us to define a reasonable starting point to use as a prototype for further tuning of the design in light of the non-functional requirements. Because the following represent only an initial implementation estimate, care must be taken to maintain as much of the flexibility designed into the system as possible. This can be done through the use of good strutured programing practices, such as the use of constants, modularization, explicit module communication, etc.

1. Implementation data structures:

-as represented in the design, using packed arrays or arrays of sequences as ARRAY

2. Disk vs main memory:

-homedoc in memory

-screendoc in memory

-all other documents on disk with direct access capability

3. Constant values:

maxvproj = 5

maxbproj = 5

The user cannot cope with too much information on the desktop at once, so the number of allowable window need not be large. Similarly the amount of information which can be shared should not be excessive, or the user will tend to lose track of it.

sdwidth = 80

sdlength = 23

As discussed earlier, the initial prototype will rebuild the screen after each user operation which changes the screen; therefore, the screendoc need not be any larger than the screen.

maxdocno = 5

maxdoclength = 40

tablelength = 120

tablewidth = 120

These limits on the documents are harsh but provide a reasonable starting point for non-functional requirement testing.

4. Speed-ups:

Prototypes reveal that some speed-up will be necessary in two areas: display creation and refreshment, and editing commands. The initial display speed-up are discussed above. Other display speed-ups will require the reuse of information gained in previous display generations. For example regeneration of only affected parts of the screendoc, screendoc larger than screen to save redefining screendoc for each small screen movement, permanent tables reflecting VPNOs or reverse video "on" for the screendoc area, etc. are potential speed-ups which will need to weighed against the space required to produce them.

Since (initially at least) all documents will be held on disk, each editing command will require one disk access and often more. For simple commands such as insert, alter, and delete, this is likely to be prohibitively slow. The natural speed-up here would be to hold part or all of the document being edited in main memory and only update the disk as required.

CHAPTER VI

Comparison of Methodologies

This chapter compares the proposed methodology against the criteria developed in Chapter II and assesses two other methodologies of interest.

VI.1 Analysis of the Case Study.

We shall begin the critique of the methodology by viewing our experience in the case study in light of the criteria previously noted. Where necessary we shall attempt to *scale up* our case study experience to allow us to at least generalize about utilizing the methodology in large system developments.

a. Against the Criteria.

Creativity and Cumulativity. In the case study, the hierarchical framework with the limitative approach provided approximately the desired degree of guidance and control. Only occasionally was there a dilemma about how to progress to the next level of design. These dilemmas were usually caused by an improper decision or design step taken earlier. In these instances backtracking was required but was greatly facilitated by the vertically structured decision levels. Certainly, experience with the design technique will remove some of the requirement for backtracking. It also appears that defining very abstract initial designs will require some experience since this is not something we have done in other design techniques. The initial design definition coupled with the horizontal decomposition is certainly the step which will require the most ability, at least until a good abstraction library becomes available.

The reapplication of abstract components (as with the editor) was very simple and saved considerable time. An unexpected benefit derived from the attempt to improve the cumulativity of software design was the ease with

which the resultant system could be adapted for variations in some of the non-functional requirements. For example, a whole spectrum of storage space/execution time relationships could have been defined with only minor changes to the constant definitions or implementation plan. This added benefit appears to result from the delaying of unnecessary decisions (like constant values) and the inclusion of redundant (derived) components within the state space. Such ability to design a family of programs in this way leads us to the possibility of using the methodology to design systems for mass production and allow minor modifications for the various clients' requirements. This would be similar to the automobile industry where the customer orders a car with one of a number of different engines, automatic or manual transmission, disk or drum brakes, etc.

Reliability. Without following the case study product through the remainder of its life, it is not possible to give absolute evidence about its reliability. However, the general point we can check is: did the methodology help to prevent errors or correct them early (ie. did the methodology reduce the cost of our required level of reliability?) *Even though our* required level of reliability did not necessitate the use of formal verification, the case study was implemented with only one minor design error being detected. (The `SIZEWINDOW` operation did not adjust the `masterspot` to coincide with the required adjustments to the `slavespot`.) This error was immediately corrected. Because of the design discipline imposed by the methodology, a number of *would-be* errors were detected before they could be propogated to lower levels of design. We agree with [Berg,82:153] that "the disciplines imposed by writing a formal specification lead to very modular and straightforward programs."

No expensive errors -- those at the system level rather than just within an algorithm -- were found. In large measure this is due to the requirement to recompose the various system components to ensure that they fit together to create the required system. In larger systems, achievement of such component compatibility (to each other and the system as a whole) will require considerably more management control than was necessary in the case study. (See the discussion on manageability below.)

The value of informal reviews of formal products (ie. at the abstract and detailed baseline design reviews) was not tested in this case study development. Since this is not a common practice, further research is required to determine its effectiveness in the enhancement of system reliability.

This is especially true since we propose to let this informal review guide at least some of the formal verification.

Requirement evolution. Evolution of functional requirements was only simulated in the case study and therefore it is difficult to make generalizations on this criterion based on the case study experience. However, the emphasis on validating the requirements in tandem with producing the design has had the beneficial effect of concentrating the design effort on the very earliest phases of the lifecycle.

The provisional user's manual as a prototype mechanism to direct the requirements evolution was very useful from the designer's viewpoint. (Again, since there was no real client, we can not comment on the client's perspective.) Although the user's manual took a significant amount of time to write, it was time well spent <u>early</u> in the design cycle. It forced consideration of a multitude of possible problems/situations and was a valuable guide in the lower levels of design. Of course, most of the effort spent on the provisional user's manual will be directly beneficial to the final user's documentation. In general, Boehm agrees with this assessment: "writing test plans and draft user's manuals tends to increase the cost of the requirements and design phases and significantly decrease the costs of testing and maintenance phases."[Boehm,81;46]

However, one general criticism has been leveled at *look ahead* techniques such as the provisional user's manual.[Horning,81] That criticism suggests that such techniques can generate designs that are unimplementable or prohibitably expensive to implement. While that danger certainly exists, the formalism and the limitative approach of this methodology helps to couteract the evolution of an unrealistic design. Additionally, the use of prototypes (Horning's proposed technique to replace look aheads) is also advocated in this methodology and can be used with the provisional user's manual to check potential implementation difficulties.

Through the Lifecycle. As can be noted in the case study, the transition from decomposed problem to the high level abstract design was greatly aided by the existence of previous abstract designs (eg. the editor). The representational transformation at the other end of design was aided by the implementation plan. Consequently, the goal of smoothing representational and phase transistions in the design has been met in this methodology.

Although the case study was not extended into the later phases of the lifecycle, the products produced in the demonstrated phases would certainly continue to be useful later in the lifecycle. The incremental design documentation with its referencing scheme should allow the maintainer to find those portions affected by proposed adaptations or corrections; while resource usage information attached to each vertical module will give him a sound basis upon which to estimate the cost of such changes.

Client involvement. The criterion of continous client involvement was not tested in the case study. The provisional user's manual would seem to be one effective method of communicating with the client. It also appears that the `Comments` and `Requirements` portions of the template along with introductory comments in the high level initial abstractions (eg. FILER) provide a very good basis for the required intermediate client documentation. The frequency and level of interaction with the client is in the purview of the design manager and is based on the ability of the client and the complexity of the system. However, there are certainly ample user interface requirements built into the process.

Manageability. One of the key factors to be assessed in the case study is whether the disciplined use of formal techniques allowed the opportunity for management control and visibility of the intermediate stages of the development -- that is, are the management and formal techniques mutually supportive?

Certainly the baselines indicated in the case project represent excellent opportunities to determine the progress and correctness of the design. The hierarchical nature of the design with decisions explicitly noted in the required documentation (the template) gives a much finer grain of detail for management control between baselines. This two level approach to the management control of the development would seem to provide sufficient opportunity for management to direct the development to the best compromise among the various development goals (including budgetary goals).

Finally, as noted earlier, it is important to insure that sufficient control can be afforded to the expensive area of formal verification. The separation of formal verification from the design effort will allow special management attention to be given to the level and cost of verification for each component.

Just as baselines and vertical level documentation provided two layers of control and visibility, the milestones and accounting information on the

template provide two layers of budgeting control. However, due to the evolutionary formulation of requirements, providing initial resource estimates may be difficult in large system developments. Only experience on a larger scale will determine if such estimates are more difficult or less accurate then initial resource estimates in other methodologies.

With no participant interactions in the case study we can only generalize about the effectiveness of communication among participants of a large project. The recording of individual decisions in both formal and informal notation (and relating the two) certainly enhances the communication of technical information about the design. The use of natural language as explanation of other portions of the formal design (eg. auxiliary definitions, operations, etc.) also has this desired effect. Because of this, the template documentation promises to provide effective communication to all participants in the design process except the client.

In the case study, we found the use of the template to be easy and effective in keeping the documentation current with the design. The only difficulty came in the last stage of design where the number of required decisions went up drastically and their individual significance fell. From this experience we suggest that there may be a point after which the explicit documentation of a number of small decisions is best left to a narrative which explains them en masse (eg. a section of the provisional user's manual.) In the stepwise refinement of any large system we can expect that at least some of the steps must be large -- usually the final ones.

One final manageability consideration is the selection of a design team structure which would best suit the communication and other management requirements of this methodology. It seems that a chief programmer teams style organization with an active manager and limited number of specialized members best matches the structure of the methodology.

b. Problems.

In addition to problems noted and recommendations made in the last section, a few other points need further consideration.

First, we need to emphasize or, in some cases, determine the limits of the methodology's usefulness. It is fairly obvious that the presented techniques are too sophisticated for use on very simple projects; however, it is not obvious just how large a suitable project must be. The methodology does not intend to guide the hardware aspects of a total system design or

software design for parallel systems. We have insufficient experience to suggest which other types of applications (if any) are not easily designed in this methodology.

Secondly, it is not yet possible for us to tell the cost of training design participants and management in the techniques of the methodology. It is obvious, however, that this methodology will be initially more difficult to learn and use than some of the non-integrated methodologies (eg. the USAF methodology.) There may be a hidden cost associated with changing an organization to the personnel structure and management organization required by this methodology. There is also some reason for concern that we are requiring too much of the project manager since we are adding a formal notation, logic, and the ability to abstract to his already substantial burden.

Thirdly, the volume of the case study documentation demonstrates that some automated support is required for the techniques to be viable in large system developments. This automation must include at least an editor and a hierarchical database system. A number of other useful automated tools are listed Section VII.2, Future Research.

Fourthly, one of the important attributes of formal design is the ability to hide previous levels of vertical decomposition for the sake of simplicity. However, this advantage becomes a disadvantage at those times when you wish to view the state space with all of its predicates and auxiliary definitions. At these times it becomes necessary to "mentally recompose" the appropriate vertical levels. Since this is strictly a textual operation, it would be very useful to have some automated assistance in vertical recomposition.

Finally, it appears likely that a number of syntactical equivalences or notational extensions would be useful in making the formalisms more understandable and easier to construct. One obvious instance of this in the case study was the specification of error conditions. A second example was the component "no change" specification in the operations. In both of these cases, we created a new *convention* to simplify the specification writing but both should be handled in a standard way in the notation. Another extension we would prefer to have in the notation is the more readable IF...THEN...ELSE... form of the alternative.

VI.2 The Hierarchical Development Methodology.

(Unless otherwise noted, the details of the Hierarchical Development Methodology are taken from [Silverberg; Robinson,78; Robinson,79].)

The Hierarchical Development Methodology of SRI International has as its goal the design and development of large scale software systems using formal techniques in an integrated approach. It is, in a number of respects, similar to the methodology we proposed in this thesis. The differences, however, are important ones and will become evident as we describe HDM.

The comprehensive approach of HDM can be explained by reviewing four aspects of the development process:

1. Structuring concepts,
2. Process guidelines and milestones (stages),
3. Languages, and
4. On-line tools.

Structuring concepts.

The structuring concepts of HDM pertain to the *development process* itself and the resultant software systems. A system in development will be decomposed (vertically) into a sequence of abstract machines, each of which provides a complete set of facilities to the next higher level. The top level machine facilities are available to the user; the lowest level (primitive machine) facilities are those provided by the environment (eg. concrete machine, operating system, high order languanges, etc.) The user machine and the primitive machine are collectively called the extreme machines.

Within each vertical level, HDM allows horizontal encapsulation (modularization) of closely related capabilities. This means that each vertical level of decomposition will have a different horizontal decomposition. Our approach was to decompose horizontally first with a different sequence of vertical levels for each horizontal module. We also consider the abstract machine to be the basic *unit of the design process*, while HDM holds the modules describing the abstract machine to be basic. However, the reasons for the horizontal decomposition are the same in both methodologies: to allow component substitution, to contain the effects of likely requirement changes, and to aid the structuring and simplicity of the design process. The different methods of hierarchical structuring indicate a significant difference in emphasis in the two methodologies which will be explained in detail later.

Process guidelines.

The guidelines included in the HDM development process are the following:

1. Decisions must be recorded as they are made. This requirement is for both a formal and an optional informal statement of the decision but no correlation is attempted between the two forms to aid readability of the formal statement. Also, the designer is encouraged to record informally any decisions he has made even if the decision will not currently be reflected in the formal specifications.

2. Decisions are to be made in favor of generality and adaptability. Parameterized abstractions, use of abstract constant representations, and judiciously postponed decisions are features of HDM which are also important in our methodology.

3. Minimize the dependency of decisions on other decisions and encapsulate related decisions into common units. Both methodologies attempt to decouple the design modules. Where the modules are not independent, HDM requires informal definition of the module interconnections within an abstract machine. In our methodology, the modules <u>are</u> the abstract machines.

4. Make explicit all sharing of decisions. In HDM, if state design decisions affect more than one module of an abstract machine, then the cross-referencing of shared decisions must be noted.

The stages.

The following stages and their associated milestones are the components of the HDM design process:

1. Conceptualization.

Conceptualization is the determination of the problem to be solved. The constraints of the system will be stated as precisely as possible in terms of user's requirements and environmental (hardware, implementation language, etc.) requirements. These requirements are stated in natural language.

This clearly demonstrates the difference in emphasis alluded to earlier between HDM and our methodology. We developed the criterion that the requirement definition must be an evolutionary process. Consequently, we advocated a very abstract initial design to simplify the translation of informal requirements and to allow explicit statement of all design decisions in the

simultaneous derivation of requirements and system design. We feel the plan to initially define not only the total requirements of the system but also the underlying environment, does not give proper importance to the requirements definition process. As we noted in Chapter II, the "precise" statement of a problem definition is often very difficult or impossible at the outset of a design process.

2. Definition of the extreme machines.

In this stage the *extreme machines* determined in the conceptualization are further defined informally using the Hierarchy Specification Language (HSL). The machines are vertically decomposed into modules, the names of the data structures and operations are listed for each module, and a description of the module interconnections are produced. The prose description conveys the decisions which were in the designer's mind as he was defining the organization of the machines.

This initial definition of the abstract user's machine and the implementation structure presupposes confidence in the problem definition set forth in the previous stage. We feel that this confidence is likely to be unfounded without a more systematic approach to the statement of the requirements, including the active involvement of the client. To define the extreme machines as required by this stage, the designer must abstractly visualize the complete user's system with all of its required decisions, he must sort the decisions by effect, and he must produce module headings for the most appropriate decomposition of the abstract machine's capabilities. This early review of the abstract design is an excellent technique for determining potentially difficult aspects of the development process; however, due to the well documented difficulty of validating informal requirements, the designer cannot be certain that these very important (costly to correct) early decisions will lead to the appropriate system.

3. Definition of the intermediate machines.

The definition of the levels of abstract machines between the extreme machines is the goal of this stage. These intermediate machines are defined in exactly the same manner as described above. The designer must determine the number of intermediate levels which is appropriate for a particular system. As noted earlier, the horizontal decomposition of each vertical level is independent. The decision about how to decompose a vertical machine level is based mainly on a concern for efficiency and algorithm design since many of the design decisions about capability of the user's machine have been made in the previous step. Therefore, the intermediate machines correspond closely to a series of refinement machines in our methodology, and the idea

of design by exclusion (or even inclusion) of capability (ie. advancement) does not exist. Now it is obvious why our hierarchical structuring differs from that of HDM -- our emphasis is on requirements definition, client interface, and structured system design, while theirs is on structured system refinement and (as we shall see) program verification.

4. Specification of modules.

This stage formalizes the decisions made in the previous two stages. Each module of every abstract machine is specified in SPECIAL. The specification includes initial values for each data structure, definition of all state transition operations, and the enumeration of objects shared between two modules. It is this stage which characterizes the functional behavior of an abstract machine. An abstract machine specification is derived by collecting the specification of its component modules.

5. Machine representations.

The fifth stage defines nonprimitive data structures in terms of data structures from the next lower level. In this way, the state of each abstact machine, including the user's machine, is defined by the data structures of the primitive machine through a series of mapping functions. These mapping functions are written in SPECIAL but are otherwise equivalent to retrieval functions between vertical levels of our design. At the end of this stage the "detailed" design is complete.

6. Abstract implementation.

The abstract implementation is used to formulate and record the final implementation decisions. An abstract program is written to implement each specified operation as a sequence of invocations of operations from the next lower level. The Intermediate Level Programming Language (ILPL) is used to describe the abstract programs.

7. Concrete implementation.

This stage creates concrete (machine executable) programs for each abstract program written in the previous stage. The ease with which this translation from abstract to concrete program is accomplished depends upon the accuracy of the initial primitive machine model. Therefore, the earliest decisions taken about the primitive machine (stages 1 and 2) will not come under close scrutiny until the end of the development process -- a potentially dangerous situation but one which occurs in some form in every methodology.

8. Verification.

In HDM, formal verification takes two forms:

1. Proofs of the properties of the user interface of the designed system (proofs associated with stages 1-4). Experience has shown that proofs of security and fault-tolerance properties can be constructed based on the HDM development approach.

2. Proof that the implementation meets its specification (proofs associated with stages 5,6, and 7). This form of verification is the motivating force behind the style of HDM software development. The techniques for implementation verification have been well defined using the Hoare pre/postcondition assertions, hierarchical structuring of the implementation, and simple abstract data assertions. Even so, the implementation verification process is so difficult that it remains an optional stage of HDM.

Languages.

Three different notations are used in HDM for different stages of the design process and a programming language is required for the system code.

The most important HDM language is SPECIAL (specification and assertion language). Based on first order predicate calculus and set theory, SPECIAL is used to define modules of the abstract machine and define the mapping relationships between data structures at successive levels. Like "Z", it supports a state-machine specification approach.

Unlike "Z", SPECIAL has a very constrained concrete syntax which is "too often awkward and unpredictable" [Silverberg;11] and the language structures (V-, O- + OV- functions) do not conform to the structure of the state-machine model. Additionally, only a subset of the language has been given formal semantics.

SPECIAL is not balanced in its abstraction capability -- procedural abstraction is much more easily developed than data abstraction. The primitive types are boolean, character, integer, and real; more complex such as records can be built into the primitive machine definition but abstract types cannot be adequately defined. Certainly there is no provision for utilizing data types from an abstraction library as we have in the case study.

Finally, the error handling mechanism built into SPECIAL will not allow a state change with an error return. With our precondition error handling technique we have not been similarly constrained.

A second generation specification language to replace SPECIAL is in design.

The Hierarchy Specification Language (HSL) is used to describe abstract machine levels and the structure of the levels. HSL specifications are used to record the sharing of decisions among modules so that consistency of their specification can be checked.

The Intermediate Level Programming Language (ILPL) describes abstract programs for each operation of every module in the design. ILPL uses data structures supplied to the modules of the primitive machine as its "built-in" data structures; therefore, it is very simple and very general.

On-line tools.

There are several tools which have been used extensively and several others which are in design. The goal is to create a development environment which is supported consistantly and thoroughly by automated support.

The first several tools are specification checkers which enforce the syntactic consistency of the SPECIAL, HSL, and ILPL specifications.

Another tool with extensive application in the Multilevel Security Verifier. This verifier is used to prove that a design specified by a set of SPECIAL modules statisfies a requirement for different levels of access security.

Tools which are still in development include a "basic" PASCAL verification system, a module simulation system for rapid prototyping prior to implementation, and a development data base system to track specification changes and do syntactic consistency checks or alert the designer of effects caused by a change.

Summary.

There are a number of basic similarities between HDM and our methodology, including the use of formalism, a hierarchical structure with abstraction and a mechanism to record design decisions. However, the differences between the two methodologies are striking. The primary divergence is one of emphasis. HDM emphasizes the later stages of the development process with a hierarchy to support the refinement of the design and the formal verification of the implementation. Only passing notice is given to the difficulties of requirements definition and validation On the other hand, we have assumed that absolute implementation verification is not currently cost effective and should be only one of the tools available to the design

manager. We also believe that a validated implementation is not of value unless it represents the solution to the clients requirements. Therefore, we have designed our methodology and organized its underlying hierarchy to support evolutionary requirement definition and validation: we emphasize the early part of the design process.

Along with this we placed great importance on the role of the client and the manager in the formal design process. These priorities are not obviously manifested in HDM although some manageability is inherent due to the incremental nature of the process. We also put an emphasis on abstraction which does not appear in HDM. It is difficult to simultaneously design large families of systems or encourage cumulativity of the design process using the Hierarchical Development Methodology.

The single greatest capability available in HDM which we are missing is early and precise expression of the interfaces between modules and between abstract machines. While this is an enviable characteristic, the price paid to obtain it in HDM is high -- the early design process should not be ignored for the sake of convenient module structuring and interface definition. However, HDM was originally created for operating system design and may be better suited for that role than our methodology because of this precision in module interface design.

VI.3 The USAF Development Methodology.

(Unless otherwise noted, the details of the USAF Development Methodology are taken from [USAF; USAFa].)

The software development methodology currently in use by the United States Air Force is predicated on the management approach to system design (see Chapter I). It was one of the first methodologies to apply the "waterfall" model of software development. The major tenets of this model are [Boehm,81:36]:

> "1. Each lifecycle phase is culminated by a verification and validation activity whose objective is to eliminate as many problems as possible in the products of that phase.
> 2. As much as possible, iterations of earlier phase products are performed in the succeeding phase."

We will review this methodology by discussing the activities and the products of each lifecycle phase. Within these phases are four baselines, six formal reviews, and two audits which provide points for management control in the development process. Standard checklists have been developed for use at these control points to assist the management of the design effort.

Documentation (communication) is given special emphasis in this methodology. Each documentory product required within the lifecycle has a predefined format and style. The standards include numbering systems and nomenclature to provide a common reference among design participants. While this rigid adherence to conventions can be constraining and even wasteful at times, it has the very beneficial effect of standardizing the communication to facilitate management and user understanding. Also, the required documentation is not constant for all developments -- certain of the documents are not required on developments which are estimated to be below standard size thresholds.

The most significant documentation required within this methodology is the specification hierarchy. It is this succession of (informal) specifications which embodies both the progressive development of the requirements and the evolution of the design. The three levels of specification are:

1. Functional Description (FD)
2. System/subsystem Specification (SS)
3. Program Specification (PS)

These documents will be explained as they are encountered within the following lifecycle description.

The Lifecycle Phases.

Concept Phase. The input to this first phase are a defined need and the technical data necessary to delineate this need. It is the objective of the concept phase to define an affordable system which is consistent with the requirements. To achieve this objective, feasibility studies (possibly including pilot programs), alternative solution comparisons, and risk assessments are accomplished. From these preliminary efforts cost estimates are produced and priorities are enumerated. If the needs can be satisfied using current technology under current budgetary contraints, then initial milestones and a schedule for development are determined.

Substantial effort is put into this initial phase to validate the need and determine the feasibility of the proposed solution. On a large project, one to four years will be allocated to the concept phase. This heavy front end loading of the lifecycle is in some respects similar to the underlying approach we have taken in our methodology; however, while we use abstraction and incremental decision making to help us define the requirements, the USAF methodology relies heavily on prototypes and simulations in order to compare alternatives.

A number of documents are produced in the phase, including the Functional Description (FD), the Data Automation Request (DAR), and the Data Project Directive (DPD). The FD is the first element in the specification hierarchy. It provides a detailed description (in the user's terminology) of the system requirements identified in the DAR. The DAR is a "conceptual description" document to transmit the proposed software project to upper management for approval. Finally, the DPD specifies management parameters and requirements for the development such as estimated resource usages, development guidelines and constraints, and project organization.

The concept phase is terminated by the functional baseline. The System Requirements Review (SRR) assesses the system requirements and accepts the FD. The concept phase checklist includes questions such as "have sufficient trade-off analyses been conducted", and "are objectives and performance envelopes defined".

Definition phase. Following the management decision to accept the requirements identified in the FD comes the definition process. This process will delineate the system to be developed. The technical requirements are allocated to individual computer program configuration items (CPCIs), the functional description is updated, and the second level of the specification

hierarchy, the system/subsystem specification (SS) is generated. The SS is a technical document which governs the development and testing of the software system. It defines the subsystem decomposition, CPCIs within each subsystem, and the interfaces among the various system parts. It also specifies the performance required of each system part. The SS represents the top level design of the required system and the customer is closely involved in its creation.

The formal acceptance of the system/subsystem specification by the System Design Review establishes the allocated baseline. The SDR will determine if the allocated requirements in the CPCIs represent a complete and optimal synthesis of system requirements, if the management concept is properly tailored to the development program ahead, and if the test plan is adequate to address the likely technical risks involved in the design. The SDR roughly correlates to the Decomposition Review in our methodology.
Development phase. The development phase begins after management ratification of the system specified in the allocated baseline. The goal of this third phase is to design, generate, and qualify the program parts (CPCIs) just defined. The development phase consists of four separate parts each of which is culminated by a review.

The first development phase part is the preliminary design. Here the initial design of CPCIs is accomplished and a test plan for each is generated. The Preliminary Design Review (PDR) brings the client and developer together to confirm the integrity of the initial design effort for each CPCI. The developer will present the results of any pilot efforts (prototypes) to the client at this time. Our Abstract Design Review corresponds to the PDR in that it seeks client approval of the initial (abstract) design of the horizontally decomposed components.

The second part of the development phase is the detailed design and includes the inital preparation of the program specification. The initial version of the program specification (PS) defines the characteristics of a computer program in sufficient detail to permit implementation in a programming language. An updated version of the PS will serve as the basic document for maintenance of the system design. The Critical Design Review satisfies the client and management that the detailed design meets its requirements and that the design is sufficiently defined to begin implementation. (This corresponds to our Detailed Design Review.)

The next portion of development phase is the implementation of the CPCIs and the development testing to debug and partially validate the CPCIs.

The Product Verification Review marks the end of this development phase part; it insures that the implementation is ready for validation testing and establishes the product baseline. The PS is frozen in this baseline.

The final portion of the development phase consists of the integration of CPCIs, the "formal" validation tests, final system documentation preparation, and system acceptance by the user. During this life cycle segment are the two required audits: the Functional Configuration Audit to verify that each CPCI has achieved the performance specified for it, and the Physical Configuration Audit to insure the system as implemented corresponds to its technical documention. The System Validation Review (SVR) completes the development phase by reviewing the qualified CPCIs and their documentation prior to transfer of system responsibility to the client and the establishment of the operational baseline.

Our Software Verification Review compares closely to the PVR of the USAF methodology. However, we have essentially combined the effects of the two audits and the SVR into our final System Review. We feel this is reasonable because the effects of the formalism will be to decrease the reliance on testing and also increase the correctness (as far as actually reflecting the system implementation) of the system documentation.

Operation phase. In the final phase of the USAF methodology the client operates the designed system based on the system documentation delivered at the operational baseline and maintains the system by reference to the program specifications. There has been no attempt in this methodology to design a family of systems or to structure design decisions so that maintenance and redesign will be simplified. Indeed, it is this very lack of emphasis on the techniques and structure of the design process (we mean design process in its narrowest sense here) that bodes very badly for the client's ability to make repairs in a cost effective manner and, even more especially, to adapt the system to changing requirements.

Summary.

Since we have used a *modified* "waterfall" model in the creation of our methodology, there are some similarities to the USAF methodology. Most notable are the use of reviews to end the significant sections of the lifecycle and the utilization of baselines and a configuration management system to *maintain control and visibility of the design products.*

The most striking difference between the two methodologies is obviously that the USAF approach does not have many software engineering features and no formal features built into it. It does not disallow the use of these concepts but it does not advocate them either. In essence then, the USAF system is just a management framework in which the business of design can be accomplished in any manner the designer deems appropriate.

Our complaint with this approach is two-fold. First, as we have seen, not using the software engineering techniques is quite likely to make the total lifecycle costs of the product much higher, even though initial production cost may be less. Secondly, if software engineering techniques, especially formal ones, are used to control the lifecycle costs, then they must be an inherent part of the management philosophy in the development as well. That is, it must be an integrated methodology not just coexistent management and formal design frameworks.

In our methodology we have placed a much greater emphasis on client involvement and have portrayed the client as much less sure of his needs. Part of this difference can be explained by the fact that the USAF methodology can expect clients from a homogeneous background with a constant level of understanding and expertise. This means that while our methodology could be used in place of the USAF approach, the reverse would not be true in many cases.

Finally, there are certainly some good lessons to be learned from the USAF methodology with its vast usage in the management of large software developments. Primary among these lessons is the need to derive and apply strict documentation conventions and review criteria (eg. checklists). Unfortunately, it is only after experience with our own methodology that we will have the necessary insight to suggest more detailed documentation conventions and management checklists.

CHAPTER VII

Conclusions

It has been the goal of this thesis to define and partially analyze by case study the rudiments of a practical methodology designed to reduce the overall lifecycle costs for large application software developments. This methodology has drawn from experience gained in the historical approaches to software design and has attempted to improve on these approaches by amalgamating selected management and software engineering techniques. Our conclusions and generalizations about the proposed methodology were presented in Section VI.1. Of course, this by no means represents the completion of this research; it is only the first phase in the methodology design.

The remainder of this chapter briefly outlines the contributions which have resulted from the research to date, with an overview of the possible future directions of related research.

VII.1 Contributions.

This study represents the first time that the set theoretical notation, "Z", has been used in the total design of a moderately sized system. It has previously been used and proven as a tool for analysis of software system architectures and as a means of documenting existing systems. In some instances we pushed the limits of this notation slightly to accommodate this new role.

To aid in the demonstration and analysis of our software methodology, we designed an information sharing, windowing display and filing system. The initial very abstract definition of the system represents a useful aid in the design and documentation of a large family of similar systems, including those with graphical capabilities.

Certainly the most significant contribution made by this research has been to make a viable start toward defining a practical system for applying simultaneously the necessary management and software engineering techniques required to solve the software dilemma.

VII.2 Future Research.

The next phase of this continuing research must be to modify the methodology based upon the conclusions drawn in Section VI.1. This *fine tuning* will prepare the way for testing in a more realistic environment; that is, with a client, limited resources, multiple designers and implementors, a management interface, etc. Only with a case study of this increased magnitude will it be possible to assess the effects of the methodology on the lifecycle and development aspects which determine system cost (eg. maintenance, reliability, productivity, user satisfaction, etc.)

Two other areas of related research are also relevant to the future viability of this methodology. The first is the creation of various automated tools to support the design process. These tools might give the capability to:

1. generate different versions of the specifications to meet the needs of the different users (implementors, designers, clients, verifiers, maintainers, etc.)

2. expand the state space definitions to explicitly show all *hidden* predicates

3. combine theories according to theory combinators

4. create/retrace the `Basis` chain to determine definitions or trace the effects of changes to the design

5. check the consistency of state predicates and types after advancement, refinement, or recomposition

6. rapidly create prototypes from specifications.

The other area of related research is in the construction of a library of high level abstractions to be used as component starting points in future designs. Such a library could also eventually prescribe standards and techniques for the combination of these components. This research would relate very closely to mode of software development seen by some (eg. [Wasserman,82]) as the future of software engineering.

ADDENDUM

Subsequent to the preparation of this thesis a very recent and substantial study of software development methodologies has come to our attention.[Wasserman,83a; Wasserman,83b] That study was done "to consolidate [past software development] work and to establish a framework from which methodologies could be developed and enhanced" -- especially with regard to Ada software projects. The result of this study was a set of general requirements for a software development methodology, a set of criteria based upon these requirements for evaluation of methodologies, and an evaluation of 24 current software methodologies. Although developed in a different style and for a different ultimate purpose, that study validates in a very large measure the criteria and the methodology set forth in this thesis. Especially important to us is the significant reinforcement that work gives to our assertion that a viable methodology must be a coherent combination of technical and management techniques spanning the whole lifecycle.

It is also interesting to note that, of the 24 current methodologies surveyed in that report, only two appear to have a formal basis which allowed some degree of formal verification. Of these two, both were still under development and had not been tested in industry. Neither addressed management issues in any detail. Obviously the need for a unified, comprehensive methodology still exists.

Appendix A

Specification Library

Notation

$\nrightarrow$ partial function

$X \nrightarrow Y =$

$\{f : X \times Y \mid (\forall y : ran(f))\ f(f^{-1}(y)) = y\}$

$\rightarrow$ total function

$X \rightarrow Y = \{f : X \nrightarrow Y \mid dom(f) = X\}$

$\rightarrowtail$ total injection

$X \rightarrowtail Y = \{f : X \nrightarrow Y \mid f; f^{-1} = dom\ (f)\} \cap X \rightarrow Y$

$(\forall a,b : X \nrightarrow Y;\ p : \mathbb{P}(X);\ q : \mathbb{P}(Y);\ v : SEQ[a];\ C : X \leftrightarrow Y)$

$\upharpoonright$ restriction

$a \upharpoonright s = a \circ \mathbb{I}(X)$

$\downharpoonright$ corestriction

$a \downharpoonright t = \mathbb{I}(Y) \circ a$

$\setminus$ domain subtraction

$a \setminus p = a \upharpoonright (X - p)$

$/$ range subtraction

$a / q = a \downharpoonright (Y - q)$

$\oplus$ functional overriding

$a \oplus b = a \setminus dom\ (b) \cup b$

$*$ closure

$a^* = \mathbb{I}(X) \cup (a^* \circ a) = \cup\{a^i \mid i : \mathbb{N}\}$

$\mapsto$ function construction

$[x \mapsto y] = \{(x,y)\}$

loop   looping

$loop(a) = I(X) \oplus (loop(a) \circ a) = a^* / dom(a)$

update   functional updating

$update(v) = v(1) \oplus v(2) \oplus \ldots \oplus v(length(v))$

uncurry $G \in (X \nrightarrow (Y \nrightarrow Z))$

$uncurry(G) \in (X,Y) \nrightarrow Z$

⦇ ⦈   image

$C⦇p⦈ = \{y : Y \mid (\exists x : p)(xCy)\}$

λ   Lambda abstraction

$(\lambda x : X)\ (term) = \{(x, term) \mid x : X\}$

τ   arbitrary element selection

μ   element creation

Schema Combinators

A schema has the form:

NAME________________________________

Signature

____

Predicates

____

'   Schema output component designator

; ∘   Schema composition

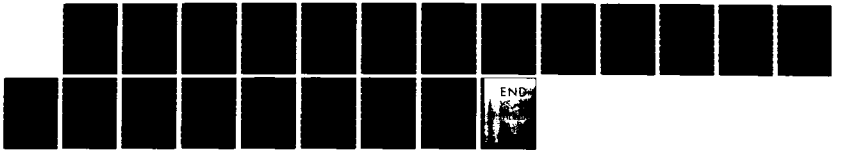
$F;G \triangleq G \circ F$

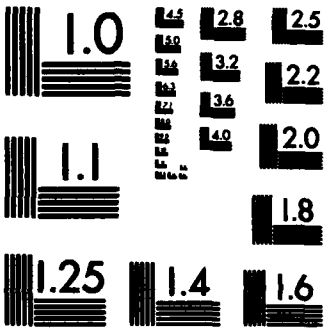
∧   Schema conjunction

Merges components in the signature; conjoins predicates.

AD-A132 569 FORMAL TECHNIQUES IN THE MANAGEMENT OF SOFTWARE DESIGN 4/4
(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH
W E RICHARDSON 17 JUN 83 AFIT/CI/NR-83-28D

UNCLASSIFIED F/G 9/2 NL





$\vee$ Schema disjunction
Merges components in the signature; disjoins predicates.

$\rhd\!\!\mapsto$ Schema component replacement
Textual substitution of schema within schema signature.

For example, given:
A, AA, and B represent schemata. P and Q are predicates.

X

A
B

P(A,B)

then:
X1 ≙ X (A $\rhd\!\!\mapsto$ AA) where AA = A | Q(A)

or expanding X1 we get

X1

AA
B

P(A,B)

=

X

A
B

P(A,B)
Q(A)

=

X | Q(A)

LOOP Schema looping

Multiple functional applications of a potentially non-homogeneous schema. In the following, F represents a schema with homogeneous (ie. both dashed and undashed) component z, non-homogeneous component a (input), and non-homogeneous component b' (output). Underlined components are sequences of the named component of F.

+

F

z : Z

$\underline{z}'$: $SEQ0_1[Z]$

$\underline{a}$: SEQ[A]

$\underline{b}'$: SEQ[B]

$\underline{z}'(0) = z$

$(\forall n : \mathbb{N}_1 \mid n < \#\underline{z}' - 1)$

$(F[z/\underline{z}'(n-1),\ z'/\underline{z}'(n),\ a/\underline{a}(n),\ b'/\underline{b}'(n)])$

LOOP

+

z' : Z

$z' = \#\underline{z}'$

A_{MODULE} Schema promotion

A is defined in the vertical level named MODULE or, if not defined there, is promoted to that level from the most recent previous explicit definition.

A[n/m] Schema component substitution

Textual replacement of term n by term m within the schema.

□ Don't care value of schema component

Θ Schema component names as an ordered tuple

Δ Delta schema
Schema with a dashed and undashed version of the named schema.

Φ Null schema
The homogeneous components of the schema is not changed by this operation. (Used in observations.)

Abstractions

1. LINE

LINE

Z

SEQ [Z]

---

2. ARRAY

ARRAY

Z

a : SEQ [1 : LINE]

(∃m : ℕ)
(∀n : dom (a))
(length (a (n)) = m)

The array is a sequence of lines which have a constant length.

3. FUNCTION

FUNCTION Z

ARRAY

length (a) = 2

(∀n,m : dom (a (1))

(a (1) (n) = a (1) (m) ⇒ n = m)

The function is an array with two lines, one for the domain and one for the range. The positional index within the line relates the two values. To be a function, each domain

value must be unique.

4. FILER

(See Chapter III.)

5. EDITOR

(See the following.)

EDITOR

Basis:

Forward: EDDOC1

Comments:

This is an extract from the display editor specification developed in [Sufrin.81b].

EDDOC1

Comments:

This component of the editor defines the *document* to be edited and the current position within that document. Several useful operations on the document are also defined.

Auxiliary Definitions:

a. speck+ is ABSTRACT

We will not define at this point what types of characters are in the document.

b. el ∈ speck+

However, we do wish to insure that an end of line marker does exist in the character set.

State Component Definition:

EDDOC1

l	:	SEQ [speck+]
r	:	SEQ [speck+]

Operations:

a.

DELETE

ΦEDDOC1

---

r ≠ ⟨⟩

l' = l

r' = tail (r)

This operation deletes the next character after the current position.

b.

TRAVERSE

⊕EDDOC1

---

r ≠ ⟨⟩
l' = l * ⟨first (r)⟩
r' = tail (r)

TRAVERSE moves the current position one character down the document.

c.

INSERT

⊕EDDOC1
c : speck+

---

l' = l * ⟨c⟩
r' = r

This operation will allow the addition of a character prior to the current position.

d.

ALTER

⊕EDDOC1
c : speck+

---

r ≠ ⟨⟩
l' = l * ⟨c⟩
r' = tail (r)

This will replace the next character and move the current position by one place.

EDITOR

State Definition:

EDITOR

new	:	EDDOC1
prior	:	EDDOC1

This will allow incorrect operations on the document to be reversed.

Operations:

a.

UNDO

ΦEDITOR

---

new' = prior
prior' = new

With this operation, changes to the document can be reversed at any time prior to the next change request. Of course, the document changing operations must be appropriately promoted to the EDITOR state.

EDDOC1A

Basis:

Prior: EDDOC1

Comments:

This refinement changes the representation of the editor document to the more familiar array form.

Auxiliary Definitions:

a. $speck \triangleq speck+ - \{el\}$

The character set required by this representation is still abstract but it no longer includes the end of line character.

b. $sp \in speck$

The space must be a legal character in this character set.

c.

$$endline : \mathbb{N}_1 \times ARRAY\ [speck] \rightarrow\!\!\!\!\!\!\rightarrow \mathbb{N}_1$$

$$endline \triangleq (\lambda l, doc \mid l \in dom\ (doc))$$
$$min\ \{b : 1..dom(doc(l))+1 \mid (\forall xx : dom(doc(l)))$$
$$(xx > b \wedge doc(l)(xx) = sp)\}$$

The end of any line in the document is defined by the first position in the line where it and all succeeding characters are blanks.

State Component Definition:

EDDOC1A

EDDOC1		
doc	:	ARRAY [speck]
position	:	PAIR [$\mathbb{N}_1$]

REFERENCES

[Abrial.80a] J.-R. Abrial
Lecture Notes on *Propositional Calculus*
M.Sc. in Computation
Oxford University, Michaelmas Term 1980.

[Abrial.80b] J.-R. Abrial
The Specification Language Z: Basic Library
Specification Group Working Paper
Programming Research Group
Oxford University, April 1980.

[Abrial.80c] J.-R. Abrial, A.A. Shuman, B. Meyer
"Specification Language"
in *On the Construction of Programs*
R.M. McKeag, A.M. Macnaghten (eds)
Cambridge University Press
New York, 1980.

[Abrial.82] J.-R. Abrial
A Theoretical Foundation to Formal Programming
unpublished
May 1982.

[Baker.72] F.T. Baker
"Chief Programmer Team Management of Production Programming"
IBM Systems Journal
Vol 11, Number 1
1972.

[Berg] H.K. Berg
"Towards a Uniform Design Methodology for
Software, Firmware, and Hardware"
in [Berg.80].

[Berg.80] H.K. Berg, W.K. Giloi (eds.)
The Use of Formal Specification Software
IFB-36
Springer-Verlag
Berlin, 1980.

[Berg.82] H.K. Berg, W.E. Boebert, W.R. Franta, T.G. Moher
Formal Methods of Program Verification and Specification
Prentice-Hall, Inc.
Engelwood Cliffs, N.J., 1973.

[Bergland.81] G.D. Bergland, R.D. Gordon (eds.)
Software Design Strategies
IEEE Computer Society Press
New York, 1982.

[Bjorner.80] D. Bjorner
"Formal Description of Programming Concepts --
a Software Engineering Viewpoint"
in *Proceedings of Mathematical Foundations of Computer Science*
Lecture Notes in Computer Science, No. 88
Springer-Verlag
Berlin, 1980.

[Bjorner.80a] D. Bjorner
Abstract Software Specifications
Lecture Note in Computer Science, No. 86
Springer-Verlag
Berlin,1980.

[Bjorner.82] D. Bjorner
Formal Specification and Software Development
Prentice-Hall International
London, 1982.

[Boebert.79] W.E. Boebert
Managing Software Projects
Seminar Notes, 1979.

[Boehm.76] B.W. Boehm
"Software Engineering"
in [Bergland.81].

[Boehm.81] B.W. Boehm
Software Engineering Economics
Prentice-Hall, Inc.
Englewood Cliffs, N.J., 1981.

[Boyd.a] D.L. Boyd, A. Pizzarello, W.T. Wood
"An Overview of RDM: Rational Design Methodology"
in [Berg.80].

[Boyd.b] D.L. Boyd, A. Pizzarello, W.T. Wood
"Abstraction and Refinement in RDM"
in [Berg.80].

[Branstad.81] M.A. Branstad, W.K. Adrian (eds)
"NBS Workshop Report on Programming Environments"
Software Engineering Notes (ACM-SIGSOFT)
Vol 6, Number 4
August 1981.

[Brooks.75] F. P. Brooks
The Mythical Man-Month: Essays on Software Engineering
Addison-Wesley
New York, 1975.

[Buckle.77] J. Buckle
Managing Software Projects
American Elsevier
New York, 1977.

[Burstall.80] R.M. Burstall, J.A. Goguen
"Semantics of CLEAR, a Specification Language"
in [Bjorner.80a]

[Burstall.81] R.M. Burstall, J.A. Goguen
"An Informal Introduction to Specifications Using CLEAR"
in *The Correctness Problem in Computer Science*
Boyer, Moore (eds.)
Academic Press
New York, 1981.

[Carlson.82] E. D. Carlson, J. R. Rhyne, D. L. Weller
"A Design for a Family of Display Management Systems"
IBM Research Report RJ3372(40408)
IBM Research Laboratory
San Jose, California, January 18,1982.

[Cave.78] W.C. Cave, A.B. Salisbury
"Controlling the Software Life Cycle -- the Project Management Task"
in [Bergland.81].

[Dahl.72] O.-J. Dahl, E.W. Dijkstra, C.A.R. Hoare
Structured Programming
Academic Press
London, 1972.

[Daniels.71] A. Daniels, D. Yeates
Systems Analysis
Science Research Associates, Inc.
California, 1971.

[Dewar.82] R.B.K. Dewar, M. Sharir, E. Weixelbaum
"Transformational Derivation of a Garbage Collection Algorithm"
Transactions on Programming Languages and Systems (ACMTOPLAS)
Vol 4, Number 4
October 1982.

[Dijkstra.68] E.W. Dijkstra
"Go To Statement Considered Harmful"
Communications of the ACM
Vol 11, Number 4
March 1968.

[Dijkstra.76] E.W. Dijkstra
A Discipline of Programming
Prentice-Hall, Inc.
Englewood Cliffs, N.J., 1976.

[Distaso.80] J.R. Distaso
"Software Management -- A Survey of the Practice in 1980"
in [Bergland.81].

[Domolki.80] B. Domolki
"An Example of Hierarchical Program Specification"
in [Bjorner.80a]

[Donaldson.78] H. Donaldson
A Guide to the Successful Management of Computer Projects
John Wiley & Sons
New York, 1978.

[Erdle.82] T.J. Erdle
PERQFILE: A Case Study in Formal Specification
Dissertation, M.Sc in Computation
Oxford University, September 1982.

[Freeman.78a] P. Freeman
"Software Design Representation: A Case Study"
Software--Practice and Experience
Vol 8
1978.

[Freeman.78b] P. Freeman
"Software Design Representation: Analysis and Improvements"
Software--Practice and Experience
Vol 8
1978.

[Gane.79] C. Gane, T. Sarson
Structured Systems Analysis: Tools and Techniques
Prentice-Hall, Inc.
Englewood Cliffs, N.J., 1979.

[Goguen.80] J. Goguen
"Thoughts on Specification, Design and Verification"
Software Engineering Notes (ACM-SIGSOFT)
Vol 5, Number 3
July 1980.

[Gries.78] D. Gries (editor)
Programming Methodology
Springer-Verlag
New York. 1978.

[Gries.81] D. Gries
The Science of Programming
Springer-Verlag
New York. 1981.

[Griffiths.78] S. N. Griffiths
"Design Methodologies in a Comparison"
in [Bergland.81]

[Guttag.82] J. Guttag. J. Horning. J. Wing
Some Remarks on Putting Formal Specifications to Productive Use (draft)
Xerox PARC
Palo Alto. California. February 1982.

[Hoare.69] C.A.R. Hoare
"An Axiomatic Basis for Computer Programming"
Communications of the ACM
Vol 12. Number 10
October 1969.

[Hoare.72] C.A.R. Hoare
"Proof of Correctness of Data Representations"
Acta Informatica
Vol 1. Number 4
1972.

[Hoare.80] C.A.R. Hoare
Lecture Notes on *Program Correctness and Validation*
M.Sc. in Computation
Oxford University. Michaelmas Term 1980.

[Hoare.81] C.A.R. Hoare
Draft Notes entitled *Screen Oriented Filing System*
Programming Research Group
Oxford University. February 1981.

[Hoare.82] C.A.R. Hoare
Programming is an Engineering Profession
Technical Monograph PRG-27
Programming Research Group
Oxford University. May 1982.

[Horning.79] J. Horning. J. Guttag
Axioms for a Display Interface, Revised (draft)
Xerox PARC
Palo Alto. California. June 1979.

[Horning.80] J. Horning, J. Guttag
Formal Specification as a Design Tool
Xerox PARC
Palo Alto, California, January 1980.

[Horning.81] J. Horning
Letter to the WG 2.3 Participants
Xerox PARC
Palo Alto, California, July 24, 1981.

[Ingrassia.78] F.S. Ingrassia
"Combating the 90% Complete Syndrome"
Datamation
January 1978.

[Jackson.75] M.A. Jackson
Principles of Program Design
Academic Press
London, 1975.

[Jensen, K.79] K. Jensen, N. Wirth
PASCAL User Manual and Report
Lecture Notes in Computer Science, No. 18
Springer-Verlag
New York, 1979.

[Jensen.79] R.W. Jensen, C.C. Tonies
Software Engineering
Prentice-Hall, Inc.
Englewood Cliffs, N.J., 1979.

[Jones.80] C.B. Jones
Software Development: a Rigorous Approach
Prentice-Hall International
London, 1980.

[Jones.81] C.B. Jones
Development Methods for Computer Programs
Including a Notion of Interference
Technical Monograph PRG-25
Programming Research Group
Oxford University, June 1981.

[Keeton.80] J. Keeton-Williams
"Needed-Verifiable Guidelines on How to Design a Methodology"
Software Engineering Notes (ACM-SIGSOFT)
Vol 5, Number 3
July 1980.

[Levitt] K.N. Levitt, L. Robinson, B.A. Silverberg
"Writing Simulatable Specifications in SPECIAL"
In [Berg.80].

[Manna.80] Z. Manna, R. Waldinger
"A Deductive Approach to Program Synthesis"
Transactions on Programming Languages and Systems (ACMTOPLAS)
Vol 2, Number 1
January 1980.

[Meyer.82] B. Meyer
"Principles of Package Design"
Communications of the ACM
Vol 25, Number 7
July 1982.

[Meyrowitz.81] N. Meyrowitz, M. Moser
"BRUWIN: An Adaptable Design Strategy for Window Manager/Virtual Terminal Systems"
Proceedings of the Eighth Symposium on Operating System Principles (ACM-SIGOPS)
Vol 15, Number 5
December 1981.

[Mallgren.82] W.R. Mallgren
"Formal Specification of Graphic Data Types"
Transactions on Programming Languages and Systems (ACMTOPLAS)
Vol 4, Number 4
October 1982.

[Morgan.82a] C. Morgan
"Specification of a Communication System"
Proceedings of an International Seminar on Synchronisation, Control and Communication in Distributed Computing Systems
Academic Press
London, 1982.

[Morgan.82b] C. Morgan
"Specification of the Cambridge Model Distributed System Nameservice"
Distributed Computed Project Working Paper
Programming Research Group
Oxford University, May 1982.

[Parker.78] J. Parker
"A Comparison of Design Methodologies"
Software Engineering Notes (ACM-SIGSOFT)
Vol 3, Number 4
October 1978.

[Parnas.72] D.L. Parnas
"On the Criteria to be Used in Decomposing Systems into Modules"
Communication of the ACM
Vol 15, Number 12
December 1972.

[Parnas.75] D.L. Parnas
"On the Design and Development of Program Families"
T. H. Darmstadt
July 2, 1975.

[Peters.77] L. J. Peters, L. L. Tripp
"Comparing Software Design Methodologies"
in [Bergland.81]

[Peters 78] L. Peters
"Relating Software Requirments and Design"
Software Engineering Notes (ACM-SIGSOFT)
Vol 3, Number 5
November 1978.

[Pressman.82] R. S. Pressman
Software Engineering: a Practitioner's Approach
McGraw-Hill Book Co.
New York, 1982.

[Pyle.81] I.C. Pyle
Towards Specifying an Information System
University of York-Department of Computer Science
Heslington, June 1981.

[Richardson.81] W.E. Richardson
Visible Filing System--A Case Study in Software Engineering
Dissertation. MSc in Computation
Oxford University, September 1981.

[Richardson.82] W. E. Richardson
"Project Simple, a Methodology Trial"
Unpublished working paper
Oxford University, November 1982.

[Riddle.78] W.E. Riddle, J.C. Wileden
"Languages for Representing Software Specifications and Designs"
Software Engineering Notes (ACM-SIGSOFT)
Vol 3, Number 4
October 1978.

[Rittel.73] H.W.J. Rittel, M.M. Webber
"Dilemmas in a General Theory of Planning"
Policy Sciences
Number 4
April 1973.

[Robinson.78] L. Robinson
HDM - Command and Staff Overview
SRI International
Menlo Park, California, February 1978.

[Robinson.79] L. Robinson, K.N. Levitt, B.A. Silverberg
The HDM Handbook, Vol. I, II, III)
SRI International
Menlo Park, California, June 1979.

[Rullo.80] T.A. Rullo (editor)
Advances in Computer Programming Management (Vol 1)
Heyden & Son, Inc.
Philadelphia, 1980.

[Scheid.80] J. Scheid
"INA JO: SDC's Formal Development Method"
Software Engineering Notes (ACM-SIGSOFT)
Vol 5, Number 3
July 1980.

[Semprevivo.76] P.C. Semprevivo
Systems Analysis: Definition, Process, and Design
Science Research Associates, Inc.
California, 1976.

[Shneiderman.80] B. Shneiderman
Software Psychology
Winthrop Publishers, Inc.
Cambridge, Mass., 1980.

[Silverberg] B.A. Silverberg
An Overview of the SRI Hierarchical Development Method
SRI International
Menlo Park, California, undated.

[Sorensen] Ib Holm Sorensen
A Design of a Display Interface
Programming Research Group
Oxford University, undated.

[Sufrin.81a] B. Sufrin
Lecture Notes on *Case Studies and Class Theory*
M.Sc. in Computation
Oxford University, Hilary Term 1981.

[Sufrin.81b] B. Sufrin
Formal Specification of a Display Editor
Technical Monograph PRG-21
Programming Research Group
Oxford University, June 1981.

[Sufrin.81c] B. Sufrin
Correctness of a Display Editor Implementation
Specification Group Working Paper
Programming Research Group
Oxford University, 1981.

[Sufrin.82] B. Sufrin
"Formal System Specification Notations and Examples"
in *Tools and Notions for Program Construction*
D. Neel (ed.)
Cambridge University Press
Cambridge, 1982.

[Swartout.82] W. Swartout, R. Balzer
"On the Inevitable Intertwining of Specification and Implementation"
Communications of the ACM
Vol 25, Number 7
July 1982.

[Tate.81] A. Tate
"High Performance Personal Computers-Circa 1980
IVCC Bulletin
Vol 3, No. 1, pg 6-10
Spring 1981.

[UCSD.79] UCSD (Mini-Micro Computer) Pascal Version II.0
Institute for Information Systems
La Jolla, California, March 1979.

[USAF] US Air Force Regulation 300-15
"Automated Data System Project Management"
January 1978.

[USAFa] US Air Force Regulation 300-12, Vol 1
"Procedures for Managing Automated Data Processing Systems"
December 1982.

[Wasserman.82] A.I. Wasserman, S. Gutz
"The Future of Programming"
Communications of the ACM
Vol. 25, Number 3
March 1982.

[Wasserman.83a] A.I. Wasserman, P. Freeman
"Ada Methodologies: Concepts and Requirements"
Software Engineering Notes (ACM-SIGSOFT)
Vol 8, Number 1
January 1983.

[Wasserman.83b] A.I. Wasserman, P. Freeman, M. Porcella
"Ada Methodology Questionaire Summary"
Software Engineering Notes (ACM-SIGSOFT)
Vol 8, Number 1
January 1983.

[Wegner.79] P. Wegner (editor)
Research Directions in Software Technology
MIT Press
Cambridge, Massachusetts, 1979.

[Weinberg.69] G.M. Weinberg
The Psychology of Computer Programming
Van Nostrand Reinhold Co.
New York, 1969.

[Wirth.71] N. Wirth
"Program Development By Stepwise Refinement"
Communications of the ACM
Vol 14, Number 4
April 1971.

[Wirth.73] N. Wirth
Systematic Programming: An Introduction
Prentice-Hall, Inc.
Englewood Cliffs, N.J., 1973

[Wirth.76] N. Wirth
Algorithms + Data Structures = Programs
Prentice-Hall, Inc.
Englewood Cliffs, N.J., 1976.

[Yourdon.78] E. Yourdon
Structured Walk-Throughs
Yourdon, Inc.
New York, 1978.

[Yourdon.79] E. Yourdon, L.L. Constantine
Structured Design
Prentice-Hall, Inc.
Englewood Cliffs, N.J., 1979.

[Zelkowitz, 79] M. Zelkowitz, A.C. Shaw, J.D. Gannon
Principles of Software Engineering and Design
Prentice-Hall, Inc.
Englewood Cliffs, N.J., 1979.

END

FILMED

9-83

DTIC